
Unicenter

TCPaccess Communications Server C/Socket Programmer Reference

Version 6.0



Computer Associates
The Software That Manages eBusiness



This documentation and related computer software program (hereinafter referred to as the "Documentation") is for the end user's informational purposes only and is subject to change or withdrawal by Computer Associates International, Inc. ("CA") at any time.

This documentation may not be copied, transferred, reproduced, disclosed or duplicated, in whole or in part, without the prior written consent of CA. This documentation is proprietary information of CA and protected by the copyright laws of the United States and international treaties.

Notwithstanding the foregoing, licensed users may print a reasonable number of copies of this documentation for their own internal use, provided that all CA copyright notices and legends are affixed to each reproduced copy. Only authorized employees, consultants, or agents of the user who are bound by the confidentiality provisions of the license for the software are permitted to have access to such copies.

This right to print copies is limited to the period during which the license for the product remains in full force and effect. Should the license terminate for any reason, it shall be the user's responsibility to return to CA the reproduced copies or to certify to CA that same have been destroyed.

To the extent permitted by applicable law, CA provides this documentation "as is" without warranty of any kind, including without limitation, any implied warranties of merchantability, fitness for a particular purpose or noninfringement. In no event will CA be liable to the end user or any third party for any loss or damage, direct or indirect, from the use of this documentation, including without limitation, lost profits, business interruption, goodwill, or lost data, even if CA is expressly advised of such loss or damage.

The use of any product referenced in this documentation and this documentation is governed by the end user's applicable license agreement.

The manufacturer of this documentation is Computer Associates International, Inc.

Provided with "Restricted Rights" as set forth in 48 C.F.R. Section 12.212, 48 C.F.R. Sections 52.227-19(c)(1) and (2) or DFARS Section 252.227-7013(c)(1)(ii) or applicable successor provisions.

© 2002 Computer Associates International, Inc. (CA)

All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Contents

Chapter 1: C Library Functions

The C Application Program Interface.....	1-2
Manipulating TPL	1-5
C Library Functions	1-6
apclose().....	1-7
apopen().....	1-8
tcheck().....	1-9
tclose().....	1-10
terror().....	1-11
texec().....	1-12
tferror().....	1-14
topen().....	1-15
tstate()	1-16
twto().....	1-17
Writing Exit Functions.....	1-18
tpl_completion_exit().....	1-19
protocol_event_exit().....	1-20
transport_provider_end_exit()	1-21
api_end_exit()	1-21
synad_error_exit()	1-22
lerad_error_exit().....	1-23
Function Prototypes.....	1-24

Chapter 2: C Language Structures

Introduction to C Language Data Structures	2-1
Correspondence Between dsects and C Language Structures.....	2-1
C Language Structures	2-2
apcb.....	2-2
apcbxl.....	2-4
tem.....	2-5
tib.....	2-6
tpa.....	2-7
tpl	2-7
tpo	2-14
tsw	2-15
tub	2-16
txl.....	2-17
txp.....	2-18

Chapter 3: Socket Library Functions

The Socket Library	3-2
BSD Sockets.....	3-3
Communication Domains and Socket Types	3-3
Constants.....	3-4
Creating Sockets and Binding Names.....	3-6
Communications Domain and Socket Type.....	3-6
Socket Descriptor	3-6
Communications Domain Name	3-7
Accepting and Initiating Connections.....	3-8
Operating in Server Mode	3-8
Operating in Client Mode.....	3-9
Once Connection is Established.....	3-9
Sending and Receiving Data	3-9
Sending Messages from Unconnected Sockets	3-9
Sending Messages from Connected Sockets	3-10
Receiving Messages from Unconnected Sockets	3-10
Receiving Messages from Connected Sockets	3-10
Sending and Receiving Messages from Noncontiguous Buffers	3-11
Using File I/O Functions	3-12
The read and write Functions	3-12
The readv and writev Functions.....	3-12
Shutting Down Connections	3-13

Socket and Protocol Options.....	3-13
Non-blocking I/O	3-14
MVS vs. UNIX	3-14
General Socket Differences	3-16
Supported Sockets.....	3-16
Endpoints	3-17
Binding a Name to a Socket.....	3-17
Urgent Data.....	3-17
Error Codes	3-18
Extra Functions.....	3-18
ANSI-C Compatible Function Prototypes	3-19
Socket Header Files.....	3-19
Options	3-20
Error Codes	3-20
UNIX File I/O Functions.....	3-21
Using UNIX Routines in MVS.....	3-21
Integrating API Socket Functions with UNIX File I/O.....	3-22
Socket Library Functions.....	3-22
Component Functional Description.....	3-23
accept().....	3-24
bind().....	3-26
close()	3-27
closelog().....	3-28
closepass().....	3-29
connect()	3-30
fcntl()	3-32
gethostbyaddr()	3-34
gethostbyname().....	3-35
gethostname()	3-37
getnetbyaddr()	3-38
getnetbyname().....	3-40
getopts.....	3-41
getpeername()	3-42
getprotobyname().....	3-43
getprotobynumber().....	3-44
getservbyname().....	3-45
getservbyport()	3-47
getsockname().....	3-48
getsockopt().....	3-49
getstablesize().....	3-55
gettimeofday().....	3-55

htonl()	3-56
htons()	3-57
inet()	3-58
inet_aton()	3-60
inet_ntoa()	3-61
ioctl()	3-62
listen()	3-64
mvselect()	3-66
openold()	3-69
ntohl()	3-71
ntohs()	3-72
openlog()	3-73
read()	3-75
readv()	3-77
perror()	3-79
recv()	3-80
recvfrom()	3-82
recvmsg()	3-85
select()	3-88
send()	3-90
sendmsg()	3-93
sendto()	3-96
setlogmask()	3-98
setsockopt()	3-99
shutdown()	3-104
socket()	3-105
sstat()	3-108
strerror()	3-109
syslog()	3-109
vsyslog()	3-110
write()	3-111
writew()	3-113

Chapter 4: Socket Library Include Files

Include File Summary	4-1
----------------------------	-----

Chapter 5: DNR Directory Services

The dirsrv() Function	5-1
Directory Services Parameter List (DPL)	5-2
Assembler Language Definition.....	5-2
C Language Definition	5-4

Chapter 6: Configuration

Socket and C Library Installation.....	6-2
Library Data Sets	6-2
Header File Library.....	6-3
Load Library	6-5
Object Library	6-5
Sample JCL.....	6-6
Socket Configuration	6-7
Configuration File	6-7
sockcfg Configuration Structure	6-7
Socket Life	6-13
Socket Buffering Limits	6-13
Compiling the Configuration File.....	6-14
Sample JCL for Compiling Socket Programs.....	6-15
IBM sockcfg Configuration Changes.....	6-15
SAS/C sockcfg Configuration Changes.....	6-17
Using Socket Libraries	6-18
Compile/Link IBM C/370 C Non-reentrant Program.....	6-18
Compile/Link IBM C/370 C Reentrant Program.....	6-19
Compile/Link SAS/C Non-reentrant Program.....	6-20
Compile/Link SAS/C Reentrant Program.....	6-21
Customizing Socket Programs	6-22

Chapter 7: UNIX System Services MVS Integrated Sockets

References	7-2
Installation Considerations	7-3
Configuration Information	7-3
Additional Socket Files.....	7-3
Additional Socket Call Parameters.....	7-4
ioctl() Parameters	7-4
getsockopt() Parameters.....	7-5
Socket-Level Options.....	7-5
TCP-Level Options	7-5
UDP-Level Options	7-6
IP-Level Options	7-6
setsockopt() Parameters	7-6
Socket-Level Options.....	7-7
TCP-Level Options	7-7
UDP-Level Options	7-7
IP-Level Options	7-8
recvmsg().....	7-8
Resolving Names and Addresses	7-8
Using /etc Files with UNIX System Services for Domain Name Resolution	7-9
Data Sets for Host Resolution.....	7-10
Search Procedures.....	7-11
Configuration	7-12
TSO Commands	7-13
CONVXL8 Command	7-13
LOADXL8 Command.....	7-14
ILATCH command	7-14
Debugging Information.....	7-15
Initialization.....	7-15
Application Issues.....	7-16
TCP SNAP ALL Command	7-16
Stopping and Starting Sockets.....	7-17
Limitations	7-17
Release Information	7-17
Common INET Support	7-18
Benefits of INET Support.....	7-18
Problems of INET Support	7-18
Problems of Multiple Physical File Systems.....	7-19

Appendix A: SAS/C Socket Library Interface (LSCNCOM)

SAS/C Socket Interface	A-1
Vendor notes on SAS/C TCP/IP socket programming:.....	A-3
Restrictions.....	A-3
Requirements.....	A-4
Certification.....	A-5
Usage.....	A-5
Using the Unicenter TCPaccess Variables	A-5
Environment Variables.....	A-6
Setup for SAS Socket /etc Files	A-7

Index

C Library Functions

This chapter describes the C Application Program Interface (API) to transport services provided by Unicenter TCPAccess and other transport providers. This chapter includes the following sections:

- [The C Application Program Interface](#) – Introduces the C Application Program Interface (API) to transport services provided by TCPAccess and other transport provider
- [Manipulating TPL](#) – Describes how to build a Transport Service Parameter List, to initiate a request, and how Unicenter TCPAccess processes the request
- [C Library Functions](#) – Provides detailed coding information for the API C library functions.
- [Writing Exit Functions](#) – Describes how to write exits for processing asynchronous events, both as normal C functions for use with the SAS/C compiler, and in assembler language for use with the IBM C/370 compiler
- [Function Prototypes](#) – Lists the function prototypes for all functions of the C library

The C Application Program Interface

The C API provides the same capability to a C language program that is available to an assembly language program that uses the API macro instructions. The description of the C interface to directory services provided by Network Directory Services (NDS) can be found in the chapter titled [DNR Directory Services](#).

The C API consists of a library of C functions that use the appropriate linkage conventions to convert from the calling sequence of the compiler to the assembler language calling conventions of the API. Support is provided for both the SAS/C and IBM C/370 compilers. The library also lets asynchronous exits be written in C. This feature is supported for use only with the SAS/C compiler.

This library provides a raw interface from an application program written in C to the assembler language macro instructions. The data structures used by the application program are the same as those used by an assembly language programmer. The C library saves no context and makes no attempt to check the validity of user parameters, or to filter errors occurring within the API. It does not generate any of its own error codes, or issue any messages. In all but a few minor cases, the C library is just a transparent pipe through which the C programmer can request service from the API, and receive the results of those requests. Thus, the user of the C library should be familiar with the API macro instructions and data structures, the request format, the types of requests, the method of returning errors, and the overall functionality of the API. Refer to *Unicenter TCPaccess Communications Server Assembler API Concepts* and *Unicenter TCPaccess Communications Server Assembler API Macro Reference* for more information.

Files Provided The C library provides the C programmer with two C include header files, and a library of object modules to be concatenated with the SYSLIB DD statement when linking the application program.

Header Files Each API data structure defined by an assembler language DSECT has a corresponding C data structure. These data structures are defined in the api.h header file. You should include this file with your compilation by coding this statement in your application program source module:

```
#include <api.h>
```

This tells both the IBM C/370 compiler and the SAS/C compiler to include the Partitioned Data Set (PDS) member API from the SYSLIB data set. Therefore, the partitioned data set containing this header file must be included in the SYSLIB DD concatenation defined during compilation. See *Unicenter TCPaccess Assembler API Concepts* for a list of standard data set names.

The api.h header file also defines manifest constants useful to application programs for making service requests and interpreting results.

ANSI C function prototype statements are provided to allow for better error checking of function calls to the C library. The prototype statements can be overridden by defining the manifest constant NOSLIBCK.

To override the ANSI C prototype statements, include this statement in the application program:

```
#define NOSLIBCK
```

The C library lets you write your API exit routines in C when using the SAS/C compiler. The header file includes function prototype statements for the various types of exit routines that demonstrate the calling conventions used. These prototype statements are commented out in the header files, since application programmers using exits provide their own function definitions with the desired function names.

Object Modules

The functions supported by the C library are provided in object module form. The library in which these modules are installed should be included in the SYSLIB concatenation that exits when the application program is link edited. The C library contains these functions:

Function	Description
apopen()	Interfaces with the AOPEN macro instruction to open an <i>Application Program Control Block (APCB)</i> and establish a session with the API subsystem. (Note: A naming conflict requires the use of apopen instead of aopen as the name of this function.)
apclose()	Interfaces with the ACLOSE macro instruction to close an APCB and release a session with the API subsystem. (Note: a naming conflict requires the use of apclose instead of aclose as the name of this function.)
tcheck()	Interfaces with the TCHECK macro instruction to check for the completion of a previous request.
tclose()	Interfaces with the TCLOSE macro instruction to close an existing endpoint, or alternatively, to pass control of the endpoint to another task or address space.
terror()	Interfaces with the TERROR macro instruction to analyze the abnormal completion of a previous request, and to generate an error message that is compatible with the WTO or WTP system macro instruction.
texec()	Interfaces to the TEXEC macro instruction to execute any arbitrary TPL-based request. Should be used to generate transport service requests that correspond to the other TPL-based macro instructions.

Function	Description
<code>tferror()</code>	A special library function that can be used to free the storage area allocated and returned by the <code>terror()</code> function (i.e., the <code>TERROR</code> macro instruction).
<code>topen()</code>	Interfaces with the <code>TOPEN</code> macro instruction to create a new endpoint, or to acquire control of an endpoint from another task or address space.
<code>tstate()</code>	Interfaces with the <code>TSTATE</code> macro instruction to obtain the current state of the endpoint.
<code>twto()</code>	A special library function that can be used to issue a system <code>WTO</code> for the error message generated by the <code>terror()</code> function (that is, the <code>TERROR</code> macro instruction).

Library functions that can be used to obtain directory services from NDS are documented in the chapter “DNR Directory Services.”

Manipulating TPL

Information is exchanged between the application program and the API in the same way that an assembler language program exchanges information. The application program builds a Transport Service Parameter List (TPL), initializes the appropriate fields, and initiates the request. The API reads the TPL, processes the request, updates the parameter list with information to be returned to the application program, and posts the TPL complete.

The format of a TPL as used by an application program written in C is exactly the same as the TPL used by an application program written in assembler language. The C structure `tpl` defined in the `api.h` header file defines the TPL for use with the C library functions. The names of fields within the TPL are the lower case equivalent of the upper case names used by the TPL DSECT.

Example

The endpoint ID, which is stored at `TPLEPID` in the assembler language DSECT, is stored at `tlepid` in the TPL C structure.

Assembler and C Language Interface Differences

The most significant difference between the assembler and C language interfaces is that the C application programmer must manipulate the TPL directly. When assembler language macro instructions are expanded at assembly time, instructions are generated to store operands in their proper locations. Generally, there is no need for an assembler language programmer to manipulate the TPL directly. Also, the keyword facility of macro instructions provides a convenient mechanism for indicating which parameters are to be manipulated.

The macro facility supported by the C language preprocessor is not nearly as robust, and is not appropriate for providing a similar interface. Also, argument passing in C is positional, and becomes unwieldy when functions may have several arguments. Therefore, the choice was made to keep the number of function arguments small and require the programmer to manipulate the TPL using the standard facilities of the language. However, C makes this relatively painless by providing a powerful grammar for dealing with data structures.

C Library Functions

This section provides detailed coding information for the API C library functions. Following a brief introductory statement summarizing its use, each function is described using the documentation style of UNIX.

C Function Components

The C library functions are presented in alphabetical order, and each function has its own section. Each page that pertains to a particular function has the name of the function in the upper outside corner.

The basic components of each function description follow:

Synopsis

A synopsis of the function is given in C language format. The function prototype statement is listed showing all function arguments, followed by declaration statements defining the format of each argument. If the function returns a value other than the normal success or failure indication, the function prototype is shown as an assignment statement.

Description

A description of the function is given, including any special rules for specifying arguments, alternative uses of the function, and any results returned.

Return Value

The function's return value, if any, is defined.

See Also

References to related functions are given.

apclose()

	Terminates session with the API subsystem.
Synopsis	<pre>#include <api.h> int apclose (apcbp) struct apcb *apcbp;</pre>
Description	<p>Terminates a session between the application program and the API and closes the APCB (Application Program Control Block) that was opened by the <code>apopen()</code> function.</p> <p>The <code>apclose()</code> function issues an <code>ACLOSE</code> macro instruction using the APCB supplied by the caller. The APCB provided as an argument of this function should be the same one provided with an earlier <code>apopen()</code> function call.</p>
Return Value	On successful completion, <code>apclose()</code> returns a 0. Otherwise, <code>apclose()</code> returns the general return code returned by the <code>ACLOSE</code> macro instruction. The user should refer to the description of the <code>ACLOSE</code> macro instruction in the <i>Assembler API Macro Reference</i> for more information on the error codes that may be returned.
See Also	<code>apopen()</code>

apopen()

Establishes a session with the API subsystem.

Synopsis

```
#include <api.h>
int apopen ( apcbp )
struct apcb *apcbp;
```

Description

Issues an AOPEN macro instruction using the APCB supplied by the calling program. The user program is responsible for initializing the appropriate fields in the APCB.

If you use the SAS/C compiler, set the apcbenvr variable to APCBSASC.

If you use the IBM C/370 compiler, set the apcbenvr variable to APCBASM.

Also, the apcbctx field of the APCB is reserved for use by the C library, and is overwritten by the apopen() function when called.

The APCB referenced by this function must be kept in storage during the life of the session, and must not be changed once the apopen() function has completed successfully. An apclose() macro instruction can be used to terminate the session, and to return the APCB to its original state.

Return Value

On successful completion, apopen() returns a zero. Otherwise, it returns the general return code returned by the AOPEN macro instruction. Refer to the description of the AOPEN macro in the *Unicenter TCPaccess Communications Server Assembler API Macro Reference* for more information on error codes.

See Also

apclose()

tcheck()

	Checks transport function status.
Synopsis	<pre>#include <api.h> int tcheck (tplp, ccodep) struct tpl *tplp; unsigned long *ccodep;</pre>
Description	<p>Checks the completion status of an active TPL by issuing the TCHECK macro instruction.</p> <p>The tcheck() function issues a TCHECK macro instruction of the form MF=(E,tplp) using the TPL pointer supplied as a function argument. If the request associated with the TPL is not complete, a system WAIT macro instruction is executed. When the TPL is posted complete, the return codes are analyzed and, if appropriate, the SYNAD or LERAD exit routine is entered. The TPL is set to inactive state.</p> <p>The TPL supplied by the caller must be in an active state. A TPL becomes active when it is used as the argument of a topen(), texec(), or tclose() function executed in asynchronous mode. If a tcheck() function is executed using an inactive TPL, an error is returned.</p>
Return Value	<p>On successful completion, tcheck() returns a zero as the function return value, error. Otherwise, it returns the value returned in register 15 by the TCHECK macro instruction. Normally, this is the general return code as defined for TPL-based macro instructions. The unsigned long pointed to by ccodep is set to the value returned by the TCHECK macro instruction in register zero. This is a conditional completion code if the TPL completed normally, or a recovery action code otherwise. The contents of register zero may also be stored in the TPL. If ccodep is the null pointer, only the function return value is returned.</p> <p>If a SYNAD or LERAD exit routine was invoked by the TCHECK macro instruction, the function return value and completion code are the values returned by the exit routine. Refer to the chapters “Socket Library Functions” and “Socket Library Include Files” for a description of information returned by the API macro instructions.</p>
See Also	tclose(), texec(), topen()

tclose()

	<p>Closes a transport endpoint.</p>
Synopsis	<pre>#include <api.h> int tclose (tplp) struct tpl *tplp;</pre>
Description	<p>Closes an endpoint, or alternatively, to pass control of the endpoint to another task or address space.</p> <p>The tclose() function issues a TCLOSE macro instruction of the form MF=(E,tplp) using the TPL pointer supplied as a function argument. Depending on the value of the TPL option code field (tplopdc3 in the union tploptcd), either the endpoint is closed and all resources allocated to the endpoint are released, or the endpoint is maintained and control is passed to the designated task or address space.</p>
Return Value	<p>On successful completion, tclose() returns a zero as the function return value, error. Otherwise, the value returned is the general return code returned by the TCLOSE macro instruction. Return codes are also stored in the TPL to provide additional information about the success or failure of the TCLOSE macro instruction.</p> <p>If asynchronous execution of the request was specified, final completion information is not returned until a tcheck() function is executed using the same TPL.</p>
See Also	<p>tcheck(), texec(), topen()</p>

terror()

	Analyzes error and generate error message.
Synopsis	<pre>#include <api.h> int terror (tplp, temp, flag) struct tpl *tplp; struct tem **temp; int flag;</pre>
Description	<p>Analyzes an error associated with a previous TPL-based function, and to generate an informative message describing the error that is suitable for displaying to the system operator or local user.</p> <p>The <code>terror()</code> function issues a <code>TERROR</code> macro instruction of the form <code>MF=(E,tplp)</code> using the TPL pointer supplied as a function argument.</p> <p>If <code>flag</code> is zero, a verbatim error message is generated. If <code>flag</code> is non-zero, a summary error message is generated.</p> <p>Information stored in the TPL by a previous request that completed abnormally is analyzed, and an error message is formatted that provides a descriptive explanation of the error. The message is suitable for displaying to a system operator or application program user, and can be output using the <code>twto()</code> function.</p>
Return Value	<p>On successful completion, <code>terror()</code> returns a zero as the function return value, error. The variable pointed to by <code>temp</code> is also updated with the address of a storage area containing the message generated by the <code>TERROR</code> macro instruction. This storage area is defined by the structure <code>tem</code>. The pointer to this structure should be supplied as an argument of <code>twto()</code> to output the message, or as an argument of <code>tferror()</code> to release the storage area. The storage area is allocated from subpool zero.</p> <p>If <code>terror()</code> fails, the function value returned is the general return code returned by the <code>TERROR</code> macro instruction. The TPL is not modified, and contains whatever information was returned by the previous request.</p>
See Also	<code>tferror()</code> , <code>twto()</code>

texec()

Executes a transport service parameter list.

Synopsis

```
#include <api.h>
int texec ( tpl, fnccd )
struct tpl *tplp;
int fnccd;
```

Description

A Transport Service Parameter List (TPL) initialized by the application program, or has been used to make a previous request, can be executed or re-executed using the texec() function. The texec() function is the library function used for making arbitrary TPL-based service requests.

The texec() function issues a TEXEC macro instruction of the form MF=(E,tplp) using the TPL pointer supplied as a function argument. If fnccd is nonzero, it must be one of the manifest constants defined in api.h corresponding to an API function code. Otherwise, the value should be zero, and the function code should have already been stored in the TPL (tplfnccd). A valid endpoint ID of an opened endpoint must also be stored in the TPL (tplepid) before the texec() function is executed.

The texec() function is used to execute all TPL-based service requests for which a C library function does not explicitly exist.

Example

To bind a protocol address to an opened endpoint, the TPL structure should be initialized with the appropriate information, and then executed using this statement:

```
texec(tplp, TFBIND)
```

This is equivalent to executing the TBIND macro instruction the following way:

```
TBIND MF=(E, tplp)
```

Which is also equivalent to this:

```
TEXEC FNCCD=TBIND, MF=(E, tplp)
```

By using the function code defined for TCLOSE, the texec() function can also be used to invoke the TCLOSE macro instruction, and this method is functionally equivalent to using tclose(). However, a special calling sequence is used to open an endpoint, therefore texec() cannot be used in place of topen(). For this reason, as well as for improved readability, it is advised that topen() and tclose() always be used to open and close endpoints.

Return Value

On successful completion, `texec()` returns a zero as the function return value, error. Depending on the actual request executed, additional information may be returned in the TPL, or in storage areas pointed to by the TPL. The application programmer should refer to the description of the corresponding TPL-based macro instruction to determine what information is returned for a particular value of `fnccd`.

When a TPL is executed in asynchronous mode, a return value of zero indicates that the service request was accepted, and the application program must wait until signaled that the request is complete. The methods used to signal the application program are the same as those defined for the API macro instructions. A `tcheck()` function must be executed to synchronize with completion of the request, schedule error recovery routines, and set the TPL inactive.

If the `texec()` function fails, the function value returned is the general return code returned by the `TEXEC` macro instruction. Normally, a specific error code and a diagnostic code, which provide additional information about the error, are returned in the TPL. The return codes that may apply to a particular function code are also listed with the description of each macro instruction. The `terror()` function may be called to generate an informative error message.

See Also

`tcheck()`, `tclose()`, `terror()`, `topen()`, `tstate()`

tferror()

	Releases transport endpoint error message.
Synopsis	<pre>#include <api.h> int tferror (temp) struct tem *temp;</pre>
Description	<p>Releases the storage allocated by the <code>terror()</code> function for returning the transport endpoint error message (<code>tem</code>) structure.</p> <p>The <code>tferror()</code> function issues a system <code>FREEMAIN</code> macro instruction to release storage containing a transport endpoint error message (<code>tem</code>). The pointer to the <code>tem</code> structure (<code>temp</code>) must have been returned by a <code>terror()</code> function, and should be supplied as a argument to <code>tferror()</code>. This storage must be released using <code>tferror()</code>, and not the generic C function, <code>free()</code>. Since each instance of <code>terror()</code> allocates a new storage area, it is advisable to execute a <code>tferror()</code> function after each successful instance of <code>terror()</code>.</p>
Return Value	On successful completion, <code>tferror()</code> returns a zero as the function return value, <code>error</code> . If the function fails, the return value is set to the return code from the <code>FREEMAIN</code> macro instruction. Once the storage area has been released, the pointer should be discarded and never again referenced.
See Also	<code>terror()</code> , <code>twto()</code>

topen()

	Opens a transport endpoint.
Synopsis	<pre>#include <api.h> int topen (tplp) struct tpl *tplp;</pre>
Description	<p>Creates an endpoint within a given communication domain and to designate the type of transport service required for the endpoint. Optionally, topen() may be used to transfer control of an endpoint to another task or address space.</p> <p>The topen() function issues a TOPEN macro instruction of the form MF=(E,tplp) using the TPL pointer supplied as a function argument. The information contained in the TPL is used to select a transport provider, and to create an endpoint within the requested communications domain. A pointer to the APCB opened by apopen() must be stored in the TPL (tplapcbp) before executing the topen() function.</p>
Return Value	<p>On successful completion, topen() returns a zero as the function return value, error. A <i>token</i> used to identify the endpoint in all future requests is also returned in the TPL (tplepid). If another TPL is used for future requests, the endpoint ID should be moved into the TPL before issuing such requests. The endpoint ID is used for addressability to the API interface routines, and must always be present.</p> <p>If topen() failed, the function value returned is the general return code returned by the TOPEN macro instruction. Additional information may be stored in the TPL to identify the particular cause of the failure.</p> <p>The topen() function can be executed in synchronous or asynchronous mode. In synchronous mode, the function value indicates the final success or failure of the request. If executed in asynchronous mode, the function value indicates whether or not the request was accepted. If so, a tcheck() function must be executed to synchronize with completion of the request, to schedule the SYNAD or LERAD exit routine, and to set the TPL inactive.</p>
See Also	tcheck(), tclose(), texec()

tstate()

	Test TPL and return endpoint state.
Synopsis	<pre>#include <api.h> int tstate (tplp, statep) struct tpl *tplp; struct tsw *statep;</pre>
Description	<p>Acquires the current state of an endpoint, and secondarily, to test if a TPL is active or incomplete.</p> <p>The tstate() function issues a TSTATE macro instruction of the form MF=(E,tplp) using the TPL pointer supplied as a function argument. If the request associated with the TPL is complete, and the TPL is inactive, the current state of the endpoint is returned in the state variable pointed to by statep. The description of the TSTATE macro instruction defines the valid states for an endpoint.</p>
Return Value	<p>On successful completion, tstate() returns a zero as the function return value, error. The location identified by statep contains endpoint state information formatted in accordance with the structure tsw.</p> <p>If the TPL is active, the tstate() function fails and the function value is set to the general return code returned by the TSTATE macro instruction. Unlike the macro instruction, which distinguishes between a complete and incomplete request, the tstate() function is only able to indicate an active TPL. No information is stored in the TPL.</p>
See Also	tcheck(), tclose(), texec(), topen()

twto()

Output error message via WTO macro instruction.

Synopsis

```
#include <api.h>

int twto ( temp, routecd, descrpcd )
struct tem *temp;
short routecd;
short descrpcd;
```

Description

Issues a WTO for the message generated by the `terror()` function. The caller can specify routing and descriptor codes to be used when issuing the WTO.

The `twto()` function issues a system WTO macro instruction to output an error message formatted by the `terror()` function. The storage area containing the message is identified by the `temp` function argument, and the default routing and descriptor codes can be modified by supplying their values with the `routecd` and `descrpcd` arguments. If either of these values is zero, the corresponding default codes are not changed. After the error message has been displayed, the storage area should be released by `tferror()`.

Return Value

On successful completion, `twto()` returns a zero as the function return value, error. If the function fails, the function value is the return code returned by the WTO macro instruction.

See Also

`terror()`, `tferror()`

Writing Exit Functions

With the C library, you can write exits for processing asynchronous events. When using the SAS/C compiler, you can write these exits as normal C functions. When using the IBM C/370 compiler, you must write the exits in assembler language.

This section describes the calling sequence for each type of exit function, and the function return values assumed by the API exit interface. If exits are required by the application program, the corresponding functions described in this section must be provided by the user of the C library. Exit routines for assembler language programs are discussed in *Planning and Operations Guide*.

Using the SAS/C
Compiler

When the SAS/C compiler is used, the APCB should be opened with the language environment specified as SAS/C. This is done by setting the `apcbenvr` variable to `APCBSASC` before executing the `apopen()` function. This instructs the API to use a language-dependent interface when scheduling exit routines.

The exit interface installs a SAS/C runtime environment in which the C-based exit function can operate properly. The environment is constructed to run without the runtime library. Consequently, many of the I/O functions of the SAS/C library cannot be used in the exit routines. The exit routines should be coded to use functions that can exist without the presence of the runtime library. If a function that requires the SAS/C runtime library is called by an exit routine, the results are unpredictable.

Using the IBM C/370
Compiler

When the IBM C/370 compiler is used, exit routines must be written in assembler language. Therefore, the APCB should be opened with the language environment specified as assembler (`apcbenvr` set to `APCBASM`). If the exit routine attempts to call a C function, it is the responsibility of the exit routine to provide the proper runtime environment. Since IBM has not exposed or documented the runtime environment used by their compiler, the API presently does not provide a built-in exit interface.

Exit Function
Descriptions

The following pages present each of the exit calls available in the Basic C Library.

tpl_completion_exit()

	TPL completion exit.
Synopsis	<pre>#include <api.h> void tpl_completion_exit (tplp) struct tpl *tplp;</pre>
Description	<p>Completes a TPL request that was issued in asynchronous mode, and specified that an exit was to be driven when the function is posted complete.</p> <p>The <code>tpl_completion_exit()</code> function is driven when a TPL request is posted complete and the associated TPL contains the address of an exit routine. The request must have been executed in asynchronous mode, and <code>TPLFEXIT</code> must have been set in <code>tplflags</code>.</p> <p>The address of the TPL that initiated the request is provided as a function argument (<code>tplp</code>). All information that is to be returned to the application program is stored before the exit routine is entered. A <code>tcheck()</code> function should be executed to set the TPL inactive, and to schedule error recovery processing, if that is appropriate.</p>
Return Value	When this exit function is complete, it should return a void value to the exit interface. Control is returned to the next sequential instruction in the C mainline program.
See Also	<code>tcheck()</code> , <code>tclose()</code> , <code>texec()</code> , <code>topen()</code>

protocol_event_exit()

	Protocol event exit.
Synopsis	<pre>#include <api.h> void protocol_event_exit (txpp) struct txp *txpp;</pre>
Description	<p>Signals a particular protocol event occurring at the local endpoint.</p> <p>The protocol_event_exit() function is driven when a particular protocol event occurs at an endpoint, and the corresponding exit routine has been defined in the APCB or endpoint exit list. These protocol events are defined:</p> <ul style="list-style-type: none">■ Connect request indication■ Connect confirm indication■ Disconnect indication■ Orderly release indication■ Normal data indication■ Expedited data indication■ Datagram error indication <p>Each of these indications has a corresponding entry in the exit list, and can have no exit routine, a unique exit routine, or an exit routine shared by other indications. All information passed to the exit routine is stored in a parameter list defined by the txp structure. A pointer to this structure (txpp) is passed as the function argument.</p>
Return Value	When this exit function is complete, it should return a void value to the exit interface. Control is returned to the next sequential instruction in the C mainline program.

transport_provider_end_exit()

	Transport provider termination exit.
Synopsis	<pre>#include <api.h> void transport_provider_end_exit (txpp) struct txp *txpp;</pre>
Description	<p>Informs the application program that the transport provider is about to end service.</p> <p>The transport_provider_end_exit() function is driven when the transport provider is being stopped or terminated. All information passed to the exit routine is stored in a parameter list defined by the txp structure. A pointer to this structure (txpp) is passed as the function argument.</p>
Return Value	When this exit function is complete, it should return a void value to the exit interface. Control is returned to the next sequential instruction in the C mainline program.
See Also	api_end_exit()

api_end_exit()

	API subsystem termination exit.
Synopsis	<pre>#include <api.h> void api_end_exit (txpp) struct txp *txpp;</pre>
Description	<p>Informs the application program that the API subsystem is about to terminate.</p> <p>The api_end_exit() function is driven when the API subsystem is being stopped or terminated. All information passed to the exit routine is stored in a parameter list defined by the txp structure. A pointer to this structure (txpp) is passed as the function argument.</p>
Return Value	When this exit function is complete, it should return a void value to the exit interface. Control is returned to the next sequential instruction in the C mainline program.
See Also	transport_provider_end_exit()

synad_error_exit()

	SYNAD synchronous error exit.
Synopsis	<pre>#include <api.h> int synad_error_exit (tplp, errp) struct tpl *tplp; unsigned long *errp;</pre>
Description	<p>Signals the occurrence of an error with a TPL-based request.</p> <p>The <code>synad_error_exit()</code> function is driven when a TPL-based request is completed with an error. This exit occurs synchronously with the execution of the user program. The TPL on which the error occurred is passed via pointer to this exit, <code>tplp</code>. When this routine processes the error, it can save a user error code in the area of storage pointed to by <code>errp</code> and also return another user-defined error as the routine's return value. The values are returned to the mainline code when a <code>tcheck()</code> function is called or as part of the synchronous completion of the TPL. The values returned by the user-supplied routine should adhere to the error number conventions of the API. Error codes are assigned to the user and these should be used by this function.</p>
Return Value	When this exit function is complete, it returns a completion code as the function return value and also another error code in the storage pointed to by <code>errp</code> .
See Also	<code>tcheck()</code> , <code>tclose()</code> , <code>texec()</code> , <code>topen()</code>

lerad_error_exit()

	LERAD synchronous error exit.
Synopsis	<pre>#include <api.h> int lerad_error_exit (tplp, errp) struct tpl *tplp; unsigned long *errp;</pre>
Description	<p>Signals a logic error occurring with a TPL-based request. This routine is identical to the <code>synad_error_exit()</code> except for the reasons this exit is driven.</p> <p>The <code>lerad_error_exit()</code> function is driven when a TPL-based request is completed with an error. This exit occurs synchronously with the execution of the user program. The TPL on which the error occurred is passed via pointer to this exit, <code>tplp</code>. When this routine processes the error, it can save a user error code in the area of storage pointed to by <code>errp</code> and also return another user-defined error as the routine's return value. The values are returned to the mainline code when a <code>tcheck()</code> function is called or as part of the synchronous completion of the TPL. The values returned by the user-supplied routine should adhere to the error number conventions of the API. There are error codes assigned to the user and these should be used by this function.</p>
Return Value	When this exit function is complete, it should return a completion code as the function return value and also another error code in the storage pointed to by <code>errp</code> .
See Also	<code>tcheck()</code> , <code>tclose()</code> , <code>texec()</code> , <code>topen()</code>

Function Prototypes

This list shows the prototypes for all functions of the C library. The function prototypes for exit function types (TPL completion exit, Protocol event exit, Transport provider end exit, Synchronous error exit, and Logic error exit) are commented out, because these are functions the C library programmer must supply if using exits.

Prototype List

These prototypes are for purposes of documentation only and show the proper calling sequence of the exit functions.

```
#if defined ( NOSLIBCK ) || defined ( unix )

extern int topen();
extern int tclose();
extern int tcheck();
extern int terror();
extern int tferror();
extern int twto();
extern int tstate();
extern int texec();
extern int apopen();
extern int apclose();
extern int dirsrv();
#else
extern int topen ( struct tpl * );
extern int tclose ( struct tpl * );
extern int tcheck ( struct tpl *, unsigned long * );
extern int terror ( struct tpl *, struct tem **, int );
extern int tferror ( struct tem * );
extern int twto ( struct tem *, short, short );
extern int tstate ( struct tpl *, struct tsw * );
extern int texec ( struct tpl *, int );
extern int apopen ( struct apcb * );
extern int apclose ( struct apcb * );
extern int dirsrv ( struct dpl * );
#endif
/* The following commented-out function headers are provided as
/* information on the user
/* routines to handle exits. These headers describe the types
/* of exit functions called and
/* the arguments passed.
/*
/* extern void tplexit ( struct tpl * );/**/ TPL completion exits*/
/* extern void protocolexit ( struct txp * );/**/ All protocol exits*/
/* extern void tpendexit ( struct txp * );/**/ TPEND exit routine*/
/* extern void apendexit ( struct txp * );/**/ APEND exit routine*/
/* extern int synadexit ( struct tpl *, unsigned long * );/**/ SYNAD exit*/
/* extern int leradexit ( struct tpl *, unsigned long * );/**/ LERAD exit*/
```

C Language Structures

This chapter provides C language definitions of TCPAccess data structures for use with the basic C library. It includes these sections:

- [Introduction to C Language Data Structures](#) – Provides a brief description of the C library data structures and shows their correspondence to C language structures
- [C Language Structures](#) – Includes the C language definition for all C language structs (structures) provided with Unicenter TCPAccess

Introduction to C Language Data Structures

Data structures that are provided by the application program as arguments of the API transport service functions are defined in this chapter and in the chapter “Socket Library Include Files.” This chapter defines these data structures as used by application programs written in C language using the basic C library.

Correspondence Between dsects and C Language Structures

There is a one-to-one correspondence between the API assembler language dsects and C language structures.

This table shows the correspondence between dsect names and structure names:

API Data Structure	DSECT Name	Structure Name
Application Program Control Block	APCB	apcb
APCB Exit List	APCBXL	apcbxl
Transport Endpoint Error Message	TEM	tem
Transport Service Information Block	TIB	tib
Transport Protocol Address	TPA	tpa
Transport Service Parameter List	TPL	tpl

API Data Structure	DSECT Name	Structure Name
Transport Protocol Options	TPO	tpo
Transport Endpoint State Word	TSW	tsw
Transport Endpoint User Block	TUB	tub
Transport Endpoint Exit List	TXL	txl
Transport Endpoint Exit Parameters	TXP	txp

Data structures are listed in detail in the following sections.

C Language Structures

The definitions and declarations in this section are written to ANSI C specifications, and can be used to compile programs with the IBM C/370 and SAS/C compilers. They are contained in the include file `api.h`.

Include files are installed on the local system as members of a partitioned data set. This data set should be included in the SYSLIB DD concatenation when the application program is compiled. The data set name is determined during installation unless changed by the local system programmer. The API member in this data set corresponds to the `api.h` include file.

Each page that pertains to a particular include file has the name of the file in the upper outside corner.

apcb

apcb (Application Program Control Block):

C Language Definition

```
/*
 * defines for length of variables used in apcb
 */
#define APCBTALN      4 /* length of control block id*/
#define APCBDGLN      2 /* length of diagnostic code */
#define APCBAMLN      4 /* length of subsys id */
#define APCBAPLN      8 /* length of application id*/
#define APCBPSLN      8 /* length of password */
/*
 * definition of the application program control block used at
 * AOPEN time
 */
struct apcb
{
```

```

char          apcbtag [ APCBTALN ]; /* control block id */
unsigned long apcbsl; /* control block length */
unsigned char apcbam; /* access method & vers */
unsigned char apcbflag; /* flag byte */
unsigned char apcboptc; /* option code */
unsigned char apcbrsv1; /* reserved */
unsigned char apcbenvr; /* language envir. code*/
unsigned char apcberrc; /* error code */
unsigned char apcbdgn [ APCBDGLN ]; /* diagnostic code */
char          *apcbamcb; /* acc. mthd control blk */
char          *apcbamcv; /* acc. mthd cvt addr. */
char          *apcbamtv; /* AM TUAS xfer vector */
char          apcbamid [ APCBAMLN ]; /* acc. mthd subsys id*/
char          *apcbrsv2; /* reserved */
struct txl   *apcbexls; /* appl lvl exit list addr */
unsigned long apcbactx; /* appl lvl context var */
unsigned long apcbectx; /* envir lvl context var */
char          apcbappl [ APCBAPLN ]; /* application id */
char          apcbpswd [ APCBPSLN ]; /* appl password */
};
/*
 * apcbtag must contain this string
 */
#define APCBIDENT "APCB"
/*
 * access method and version ( apcbam )
 */
#define APCBAMSK 0xF0 /* access method id */
#define APCBATLI 0x10 /* transport layer interface */
#define APCBAMAX APCBATLI /* maximum access method */
#define APCBAVER 0x0F /* access method version */
/*
 * flag byte ( apcbflag )
 */
#define APCBFSTP 0x80 /* applid is stpnam frm tiot */
#define APCBF31B 0x40 /* AMODE=31 */
#define APCBFANY 0x20 /* RMODE=ANY */
#define APCBFOPN 0x10 /* apcb is open */
#define APCBFERR 0x08 /* permanent error flag */
#define APCBFTRM 0x04 /* task termina. in progress */
#define APCBFBSY 0x01 /* open/close in progress */
/*
 * option code ( apcboptc )
 */
#define APCBOTRC 0x80 /* optcd=notrace | trace */
#define APCBOGTF 0x40 /* optcd=nogtf|gtf */
/*
 * language environment code ( apcbenvr )
 */
#define APCBASM 0 /* assembler language */
#define APCBIBMC 1 /* ibm c */
#define APCBSASC 2 /* sas c */
#define APCBPLI 3 /* pli */
#define APCBCOBL 4 /* cobol */
#define APCBFORT 5 /* fortran */
#define APCBEMAX APCBFORT /* maximum environment code */
/*
 * error code ( apcberrc )
 */
#define APCBECFG 1 /* subsystem not configured */
#define APCBEACT 2 /* subsystem not active */
#define APCBERDY 3 /* subsystem not initialized */
#define APCBESTP 4 /* subsystem is stopping */
#define APCBEDRA 5 /* subsystem is draining */
#define APCBEVCK 6 /* apcb validity check error */
#define APCBELER 7 /* internal logic error */

```

```
#define APCBEPRB      8      /* not issued from PRB */
#define APCBEOPN     9      /* apcb already opened */
#define APCBECLS    10      /* apcb already closed */
#define APCBEBSY    11      /* apcb busy w/aopen/aclose */
#define APCBEPER    12      /* apcb has permanent error */
#define APCBECVT    13      /* access mthd cvt not avail */
#define APCBEMEM    14      /* insuff. memory avail */
#define APCBEENV    15      /* cannot initialize enviro */
#define APCBEBEG    16      /* cannot estab api session */
#define APCBEVER    17      /* invalid access mthd vers */
#define APCBEOPT    18      /* invalid/unsupported opt */
#define APCBEDUP    19      /* dup session for this am */
#define APCBEAMD    20      /* AMODE inconsist w/AOPEN */
#define APCBETRV    21      /* AMTV validity check error */
#define APCBEEND    22      /* Cannot release API sess */
#define APCBERMX    APCBEEND /* max apcb error code */
#define APCBLEN     sizeof(apcb) /* length of apcb */
```

apcbxl

apcbxl (Application Program Control Block Exit List):

C Language Definition

```
/*
 * definition of the application program control block exit list
 */
struct apcbxl
{
    unsigned long    apcbxlen;    /* total len of exit list */
    struct txl      *apcbxlst;    /* exit rtn entry pts list */
};
```

tem

tem (Transport Endpoint Error Message):

C Language Definition

```
#define TEMSGTAGLEN    4          /* length of control block */
#define TEMSGIDLEN     8          /* length of message ID   */
#define TEMSGBDYLEN   26         /* length of message body */
#define TEMSGLEN      34         /* length of message      */
/*
 * terror generated message
 */
union temmsg
{
    char    temmsgtxt [ TEMSGLEN ];      /* msg txt len (1st)*/
    struct
    {
        char    temsgid [ TEMSGIDLEN ]; /* message id */
        char    temsgbdy [ TEMSGBDYLEN ]; /* message body */
    } temsparts;
};
/*
 * terror generated message parameter list return value
 *
 * a pointer to this structure is returned when a terror()
 * function completes successfully.
 */
struct tem
{
    char    temtag[ TEMSGTAGLEN ];      /* cntrl block tag */
    unsigned long temsl;                /* subpool and length*/
    char    temsglen [2];              /* msg length + 4 (1st line) */
    char    temmcsf1;                  /* mcs flag byte #1 */
    char    temmcsf2;                  /* mcs flag byte #2 */
    union temmsg temmsg;               /* first line */
    unsigned short temdesc;            /* descriptor codes */
    unsigned short temrout;            /* routing codes */
    char    temsgtyp [2];              /* mlwto line typ (1st line) */
    char    temarea;                  /* mlwto area id */
    char    temnline;                 /* mlwto number of lines */
    short   temmlen;                  /* mlwto line length + 4 */
    short   temmltyp;                 /* mlwto line type */
    char    temmltxt;                 /* mlwto line text */
                                        /* element size is variable */
};
#define TEMLEN          sizeof ( struct tem )
```

tib

tib (Transport Service Information Block):

C Language Definition

```
#define TIBSYSIDLEN      4    /* max length of subsystem name */
#define TIBSVCIDLEN     8    /* max length of service name */
#define TIBHOSTNAMELEN 64   /*max length of local host name */
/*
 * structure returned by tinfo()function
 */
struct tib
{
    unsigned char  tibtsdom;    /* transport service domain */
    unsigned char  tibststyp;  /* transport service type */
    unsigned char  tibtschr;   /* transport service char. */
    unsigned char  tibtsopt;   /* transport service options */
    char           tibsysid [ TIBSYSIDLEN ]; /* TP subsys name */
    char           tibsvcid [ TIBSVCIDLEN ]; /* TP service name */
    int            tibproto;   /* transport protocol number */
    int            tibqlstn;   /* max size of listen queue */
    int            tibqsend;   /* max size of send queue */
    int            tibqrecv;   /* max size of receive queue */
    int            tibltsnd;   /* max size of send TIDU */
    int            tibltrcv;   /* max size of receive TIDU */
    int            tiblsend;   /* max size of send buffer*/
    int            tiblrecv;   /* max size of rcv buffer */
    int            tibladdr;   /* max size of protocol address */
    int            tibloptn;   /* max size of protocol options */
    int            tibltsdu;   /* max size of TSDU */
    int            tiblxpdt;   /* max size of ETSDU */
    int            tiblconn;   /* max size of connect data */
    int            tibldisc;   /* max size of disconnect data */
    int            tiblinfo;   /* max size of information unit */
};
/*
 * defines for transport service domain
 */
#define TIBDINET        2    /* Internet domain */
#define TIBDACP         4    /* ACP inet domain */
/*
 * transport service type
 */
#define TIBTCOTS        1    /* connection-mode service */
#define TIBTCLTS        2    /* connectionless-mode service */
/*
 * transport service characteristics
 */
#define TIBCTSDU        0x10 /* message boundaries preserved*/
#define TIBCXPD        0x08 /* expedited data supported */
#define TIBCOPTN        0x04 /* user-settable options supported */
#define TIBCCOND        0x02 /* connect with user data */
#define TIBCDISD        0x01 /* disconnect with user data */
/*
```

tpa

```

* transport service options
*/
#define TIBOASSO    0x80 /* datagram associations supported */
#define TIBOSCND   0x40 /* secondary information available */
#define TIBOSTAT   0x20 /* statistical info available */
#define TIBCRLSE   0x10 /* orderly release supported */
/*
* length of the TIB
*/
#define TIBLEN     sizeof ( struct tib )

```

tpa (Transport Protocol Address):

C Language Definition

```

/*
* transport protocol address in the internet domain
*/
struct tpainet
{
    unsigned short    tpainetd;    /* internet domain */
    unsigned short    tpainett;    /* TCP port number */
    unsigned long     tpainetn;    /* IP host address */
};
/*
* length of transport protocol address
*/
#define TPALEN     sizeof ( struct tpainet )

```

tpl (Transport Service Parameter List):

C Language Definition

```

#define TPLSVCIDLEN 8
/*
* internal ECB, external ECB address, exit routine address union
*/
union tplecbexit
{
    union
    {
        unsigned long tpliercb; /* internal ECB */
        unsigned long *tplxpcb; /* external ECB address */
    } tplecb;
    void (*tplexit)(); /* exit routine address */
};
/*
* TPL option code structure
*/
union tploptcd
{
    unsigned long    tploptcdl;    /* option codes */
    struct
    {
        unsigned char    tplopcd1;    /* option code #1 */
        unsigned char    tplopcd2;    /* option code #2 */
        unsigned char    tplopcd3;    /* option code #2 */
        unsigned char    tplopcd4;    /* option code #2 */
    } tplopcds;
};

```

```
/*
 * TPL return code structure
 */
union tplrtncd
{
    unsigned long    tplrtncdl;        /* return codes */
    struct
    {
        unsigned char    tplactcd;    /* recovery action code*/
        unsigned char    tplerrcd;    /* specific error code */
        unsigned short    tpldgncd;   /* diag & sense codes */
    } tplrtncds;
};
/*
 * TPL fixed-length parameters structure
 */
struct tplparm
{
    union
    {
        int            tplqlstn;      /* listen queue length */
        int            tplseqno;      /* sequence number */
        unsigned char *tpltcb;        /* tcb address */
    } tplparm1;
    union
    {
        unsigned long    tplnewep;    /* new endpoint */
        unsigned char *tplascb;       /* ascb address */
        int            tplcount;      /* transfer byte count */
    } tplparm2;
    union
    {
        unsigned char *tpluser;       /* tub or acee address */
        unsigned long    tpldiscd;    /* disconnect reason code */
        unsigned long    tpldgerr;    /* datagram error code */
        unsigned long    tplstate;    /* old endpoint state */
        unsigned long    tplxcnt;     /* xdata residual count */
    } tplparm3;
};
/*
 * TPL variable-length protocol address parameter
 */
struct tpladdr
{
    unsigned char *tpladbuf;          /* parameter address */
    int            tpladlen;          /* parameter length */
};
/*
 * TPL variable-length user data parameter
 */
struct tpldata
{
    unsigned char *tpldabuf;          /* parameter address */
    int            tpldalen;          /* parameter length */
};
/*
 * TPL variable-length protocol options parameter
 */
struct tploptn
{
    unsigned char *tplopbuf;          /* parameter address */
    int            tploplen;          /* parameter length */
};
/*
 * TPL variable-length parameters structure
 */
```

```

struct tplbufp
{
    struct tpladdr    tpladdr;    /* protocol address */
    struct tpldata    tpldata;    /* user data */
    struct tploptn    tploptn;    /* protocol options */
};
/*
 * TPL structure specific to TOPEN
 */
struct tplopen
{
    unsigned char    tpldom;        /* communication domain */
    unsigned char    tploflag;    /* open flags */
    union
    {
        unsigned short    tpltype;    /* transport serv type */
        unsigned short    tplproto;    /* transport proto num */
    } tplservc;
    struct apcb    *tplapcbp;    /* address of APCB */
    char    tplsvcid [ TPLSVCIDLEN ]; /* TP service name */
    struct txlh    *tplexlst;    /* address of exit list */
    unsigned long    tplcntx;    /* word of user context */
};
/*
 * Transport Service Parameter List (TPL) structure
 */
struct tpl
{
    unsigned char    tplident;    /* control block identifier */
    unsigned char    tplfnccd;    /* function code */
    unsigned char    tplactiv;    /* active semaphore */
    unsigned char    tplflags;    /* flags used by AP */
    union
    {
        unsigned long    tplepid;    /* endpoint id */
        unsigned long    tpltcep;    /* TCEP address */
    } tlep;
    union tplecbexit    tplecbexit; /* ECB or exit routine addr */
    union tploptcd    tploptcd;    /* option codes */
    union tplrtncd    tplrtncd;    /* return codes */
    struct tplparm    tplparm;    /* fixed-length parameters */
    union
    {
        struct tplbufp    tplbufp    /* variable-length parms */
        struct tplopen    tplopen;    /* tpl open parameters */
    } tplobbfp;
};
/*
 * control block ID values (tplident)
 */
#define TPLIDSTD 0xEA                /* standard long format */
#define TPLIDSHT 0xEB                /* short format */
/*
 * defines for tpl function codes ( tplfnccd )
 */
#define TFORG1    0    /* origin for std functions */
#define TFACCEPT    1    /* accept connection request */
#define TFADDR    2    /* get address information */
#define TFBIND    3    /* bind protocol address */
#define TFCLEAR    4    /* confirm disconnect*/
#define TFCLOSE    5    /* close endpoint */
#define TFCONFRM    6    /* confirm connection req */
#define TFCONNCT    7    /* initiate connection req */
#define TFDISCON    8    /* initiate disconnect */
#define TFINFO    9    /* get transport service info*/
#define TFLISTEN    10    /* listen for connection req */

```

```

#define TFOPEN          11      /* open an endpoint */
#define TFOPTION        12      /* negotiate options */
#define TFRECV          13      /* receive data */
#define TFRECVERR       14      /* receive datagram error */
#define TFRECVFR        15      /* receive datagram */
#define TFREJECT        16      /* reject connection request */
#define TFRELACK        17      /* confirm orderly release */
#define TFRELESE        18      /* initiate orderly release */
#define TFRETRCT        19      /* retract a listen request */
#define TFSEND          20      /* send data */
#define TFSENDTO        21      /* send datagram */
#define TFUNBIND        22      /* unbind protocol address */
#define TFUSER          23      /* associate user id */
#define TFMAX1          TFUSER
#define TFORG2          128     /* origin for ctl functions */
#define TFCHECK         129     /* check tpl for completion */
#define TFERROR         130     /* format error message */
#define TFSTATE         131     /* get endpoint state */
#define TFMAX2          TFSTATE
/*
 * flag byte of tpl defines ( tplflags )
 */
#define TPLFCMPL        0x80     /* tpl completed */
#define TPLFCERR        0x40     /* tpl completed with error */
#define TPLFXECB        0x20     /* external ecb being used */
#define TPLFEXIT        0x10     /* exit being used */
#define TPLF31B         0x08     /* tpl issued in 31 bit mode */
#define TPLFACPT        0x04     /* accepting on this endpoint */
/*
 * option code #1 defines ( tlopdc1 )
 */
#define TOASYNC         0x80     /* OPTCD=SYNC | ASYNC */
#define TOSHORT         0x40     /* OPTCD=LONG | SHORT */
#define TOTRUNC         0x20     /* OPTCD=NOTRUNC | TRUNC */
#define TONEGOT         0x10     /* OPTCD=NONEGOT | NEGOT */
#define TONOBLOK        0x04     /* OPTCD=BLOCK | NOBLOCK */
/*
 * option code #2 defines ( tlopdc2 )
 */
#define TOMORE          0x80     /* OPTCD=NOMORE | MORE */
#define TOEXPDTE        0x40     /* OPTCD=NORMAL | EXPEDITE */
#define TONOTEOM        0x20     /* OPTCD=EOM | NOEOM */
#define TOABORT         0x10     /* OPTCD=CLEAR | ABORT */
#define TOINDIR         0x08     /* OPTCD=DIRECT | INDIR */
#define TODLOCAL        0x04     /* OPTCD=DNOTLOCAL | DLOCAL */
#define TOLOCATE        0x02     /* OPTCD=NOLOCATE | LOCATE */
/*
 * option code #3 defines ( tlopdc3 )
 */
#define TOACEE          0x80     /* OPTCD=TUB | ACEE */
#define TOCIPHER         0x40     /* OPTCD=PLAIN | CIPHER */
#define TOOLD           0x20     /* OPTCD=NEW | OLD */
#define TOASSIGN        0x10     /* OPTCD=USE | ASSIGN */
#define TOREMOTE        0x08     /* OPTCD=LOCAL | REMOTE */
#define TOPASS          0x04     /* OPTCD=DELETE | PASS */
/*
 * option code #4 defines ( tlopdc4 )
 */
#define TOINFO          0xC0     /* tinfo option codes */
#define TOPRIMRY        0x00     /* OPTCD=PRIMARY */
#define TOSCNDRY        0x80     /* OPTCD=SECNDRY */
#define TOSTATS         0x40     /* OPTCD=STATS */
#define TOOPTION        0x30     /* toption option codes */
#define TODECLAR        0x00     /* OPTCD=DECLARE */
#define TOVERIFY        0x10     /* OPTCD=VERIFY */
#define TOQUERY         0x20     /* OPTCD=QUERY */

```

```

#define TODFAULT      0x30      /* OPTCD=DFAULT */
#define TOAPI         0x08      /* OPTCD=TP | API*/
/*
* recovery action codes ( tplactcd )
*/
#define TAOKAY        0         /* successful completion*/
#define TAEXCPTN      4         /* exceptional condition */
#define TAINTEG       8         /* connection/data integrity error */
#define TAENVIRO      12        /* environmental condition */
#define TAFORMAT      16        /* format or specification error */
#define TAPROCED      20        /* sequence or procedure error */
#define TATPLERR      24        /* logic errors with no tpl rtncd */
#define TAUSER        28        /* user-defined action codes */
/*
* conditional completion code ( tplerrcd )
*/
#define TCOKAY        0x00      /* 00: no conditionals */
#define TCVERIFY      0x80      /* 00: options did not verify */
#define TCNEGOT       0x40      /* 00: options negotiated */
#define TCTRUNC       0x20      /* 00: buffer truncated */
#define TCSTOP        0x08      /* 00: subsystem is stopping */
/*
* specific error code ( tplerrcd )
*/
/* TENOEP was made obsolete and replaced by TEDRAIN */
/* #define TENOEP      1         /* 04: no new endpoints allowed */
#define TENONEGO      6         /* 04: no negotiation allowed */
#define TENOBLOK      9         /* 04: no blocking allowed */
#define TENOLSTN      10        /* 04: no listen pending */
#define TEPROTO       1         /* 08: protocol error */
#define TEOVRFLO      2         /* 08: buffer overflow */
#define TEDISCON      3         /* 08: disconnect received */
#define TERELEASE     4         /* 08: orderly release received */
#define TEOVLAY       5         /* 08: control block overlaid */
#define TEFLOW        9         /* 08: temporary flow control */
#define TERETRCT      10        /* 08: listen retracted */
#define TEPURGED      11        /* 08: request purged for TCLOSE */
#define TESYSERR      1         /* 12: system error */
#define TESUBSYS      2         /* 12: subsystem error */
#define TENOTCNF      3         /* 12: subsys not config in o/s */
#define TENOTACT      4         /* 12: subsys not started */
#define TENOTRDY      5         /* 12: subsys not initialized */
#define TEDRAIN       6         /* 12: subsys drained by operator */
#define TESTOP        7         /* 12: subsystem stopped by open. */
#define TETERM        8         /* 12: subsys abnormally termin */
#define TEUNSUPO      9         /* 12: unsupported option/facility */
#define TEUNSUPF     10         /* 12: unsupported function/serv */
#define TEUNAVBL     11         /* 12: unavailable service/facil */
#define TEUNAUTH     12         /* 12: user unauthorized */
#define TERSOURC     13         /* 12: insufficient resources */
#define TEINUSE      14         /* 12: TPA in use */
#define TEBDOPCD      1         /* 16: invalid option code */
#define TEBDEPID      2         /* 16: invalid endpoint */
#define TEBDXECB      3         /* 16: invalid exit/ecb address */
#define TEBDDOM       4         /* 16: invalid communication dom */
#define TEBDPROT      5         /* 16: invalid transport protocol*/
#define TEBDTYPE      6         /* 16: invalid transport serv type */
#define TEBDXLST      7         /* 16: invalid exit list */
#define TEBDUSER      8         /* 16: invalid user parm*/
#define TEBDACEE      9         /* 16: invalid accessor element */
#define TEBDSQNO     10         /* 16: invalid sequence number */
#define TEBDQLEN     11         /* 16: invalid queue length */
#define TEBDTCB      12         /* 16: invalid tcb address */
#define TEBDASCB     13         /* 16: invalid ascb address */
#define TEBDADDR     14         /* 16: invalid protocol address */
#define TEBDOPTN     15         /* 16: invalid options */

```

```

#define TEBDDATA      16      /* 16: invalid data buffer */
#define TEBDTSID     18      /* 16: invalid transport serv id */
#define TESTATE      1       /* 20: invalid state for function */
#define TEINEXIT     2       /* 20: invalid function w/in exit */
#define TEINACTV     3       /* 20: check issued to inact tpl */
#define TEINCMPL     4       /* 20: endpoint has incomplete fnc */
#define TEINDICA     5       /* 20: pending connect indication */
#define TEBUFOVR     6       /* 20: send / rcv buffer overrun */
#define TEREQOVR     7       /* 20: send / rcv request overrun */
#define TENOCONN     8       /* 20: no connection */
#define TENODISC     9       /* 20: no disconnect indication */
#define TEOUTSEQ     10      /* 20: request is out of sequence */
#define TENOERR      11      /* 20: no error indication */
#define TEAMODE      13      /* 20: AMODE conflicts with APCB */
#define TEOWNER      14      /* 20: not opened by this task */
#define TELISTEN     15      /* 20: listen q full */
#define TEACCEPT     16      /* 20: accepting on this endpoint */
#define TEB4EXIT     1       /* 24: TPL check before exit */
#define TEACTIVE     2       /* 24: TPL is still active */
#define TEMAXCODE    20      /* MAX error code defined */
/*
 * disconnect reasons codes
 */
#define TDTRANTO     1        /* Transmission timeout */
#define TDHOSTUN     2        /* Host unreachable */
#define TDPORTUN     3        /* Port unreachable */
#define TDRABORT     4        /* Remote aborted connection */
#define TDLNIOWN     5        /* Local net I/F down */
#define TDPROTUN     6        /* Protocol unreachable */
#define TDACPRR      7        /* ACP connection error */
#define TDAPIRR      8        /* API connection error */
/*
 * minimum TPL length
 */
#define TPLMIN (sizeof (unsigned char) + sizeof (unsigned char) + \
               sizeof (unsigned char) + sizeof (unsigned char) + \
               sizeof (unsigned long) + sizeof (struct tplecbexit) + \
               sizeof (struct tploptcd)+ sizeof (struct tplrtncd) )
/*
 * length of short form TPL ( function-specific )
 */
#define TLRELACK     TPLMIN
#define TLRELESE     TPLMIN
#define TLUNBIND     TPLMIN
#define TLRETRCT     TPLMIN
#define TLACCEPT     ( TPLMIN + sizeof ( struct tplparm ) )
#define TLCLEAR      ( TPLMIN + sizeof ( struct tplparm ) )
#define TLCLOSE      ( TPLMIN + sizeof ( struct tplparm ) )
#define TLDISCON     ( TPLMIN + sizeof ( struct tplparm ) )
#define TLREJECT     ( TPLMIN + sizeof ( struct tplparm ) )
#define TLUSER       ( TPLMIN + sizeof ( struct tplparm ) )
#define TLADDR       ( TPLMIN + sizeof ( struct tplparm ) + \
                       sizeof ( struct tpladdr ) )
#define TLBIND       ( TPLMIN + sizeof ( struct tplparm ) + \
                       sizeof ( struct tpladdr ) )
#define TLCONFRM     ( TPLMIN + sizeof ( struct tplparm ) + \
                       sizeof ( struct tpladdr ) )
#define TLCONNCT     ( TPLMIN + sizeof ( struct tplparm ) + \
                       sizeof ( struct tpladdr ) )
#define TLLISTEN     ( TPLMIN + sizeof ( struct tplparm ) + \
                       sizeof ( struct tpladdr ) )
#define TLRECVER     ( TPLMIN + sizeof ( struct tplparm ) + \
                       sizeof ( struct tpladdr ) )
#define TLINFO       ( TPLMIN + sizeof ( struct tplparm ) + \
                       sizeof ( struct tpladdr ) + \
                       sizeof ( struct tpldata ) )

```

```

#define TLRECV      ( TPLMIN + sizeof ( struct tplparm ) + \
                    sizeof ( struct tpladdr ) + \
                    sizeof ( struct tpldata ) )
#define TLRECVFR   ( TPLMIN + sizeof ( struct tplparm ) + \
                    sizeof ( struct tpladdr ) + \
                    sizeof ( struct tpldata ) )
#define TLSEND     ( TPLMIN + sizeof ( struct tplparm ) + \
                    sizeof ( struct tpladdr ) + \
                    sizeof ( struct tpldata ) )
#define TLSENDTO   ( TPLMIN + sizeof ( struct tplparm ) + \
                    sizeof ( struct tpladdr ) + \
                    sizeof ( struct tpldata ) )
#define TLOPTION   ( TPLMIN + sizeof ( struct tplparm ) + \
                    sizeof ( struct tpladdr ) + \
                    sizeof ( struct tpldata ) + \
                    sizeof ( struct tploptn ) )

/*
 * length of standard ( long ) form TPL
 */
#define TPLEN      ( sizeof ( struct tpl ) + \
                    sizeof ( struct tplbufp ) - \
                    sizeof ( struct tplopen ) )
#define TLOPEN     sizeof ( struct tpl )
#define TPLMAX     sizeof ( struct tpl )

/*
 * general return codes ( returned in R15 )
 */
#define TROKAY     0    /* successful cml, accepted */
#define TRFAILED  4    /* unsuccessful cml, not accepted */
#define TRFATLFC  8    /* invalid function code */
#define TRFATLPL  12   /* fatal tpl error */
#define TRFATLAM  16   /* fatal access method error */
#define TRFATLAP  20   /* apcb is closed */
#define TRUSER    24   /* first user return code */

/*
 * topen communication domains
 */
#define TDINET     2    /* Internet domain */
#define TDACP      4    /* ACP internet domain */
#define TDMAX     TDSNA /* max value for domain */

/*
 * topen open flags
 */
#define TPLOFPRO   0x80 /* protocol number specified */
#define TPLOFORD   0x40 /* COTS orderly release required */
#define TPLOFASO   0x20 /* CLTS association required */

/*
 * topen transport service types
 */
#define TTCOTS     1    /* connection-mode service */
#define TTCLTS    2    /* connectionless-mode service */
#define TTMAX     TTCLTS /* max value for service type */

/*
 * topen transport protocol number
 */
#define TPINTTCP   6    /* darpa internet tcp */
#define TPINTUDP  17   /* darpa internet udp */
#define TISOTP4   0    /* iso transport class 4 */
#define TLOPEN    sizeof ( struct tpl )
                    /* length of short tpl: topen */
#define TPLMAX    sizeof ( struct tpl )
                    /* maximum tpl length: topen */

```

tpo

tpo (Transport Protocol Options):

C Language Definition

```

/*****
/* This structure defines the format of an option when          */
/* issuing a TOPTION request                                  */
/*****
#define TPOMAXLEN      320          /* for ifconfig */
struct tpo
{
    short          tpoptlen          /* option length */
    short          tpooption;        /* option name */
    unsigned char  tpovalue [ TPOMAXLEN ]; /* option value */
};
/*
 * API defined option names
 */
#define TPOAQSND      0          /* max # of sends */
#define TPOAQRVC      1          /* max # of recvs */
#define TPOALSND      2          /* length of send buffer */
#define TPOALRCV      3          /* length of receive buf */
/*
 * ACP defined option names
 */
#define TPOPRWND      1          /* TCP receive window */
#define TPOPKTIM      2          /* TCP Keepalive time */
#define TPOPKKEEP      3          /* TCP Keepalive options */
#define TPOPDNAG      4          /* Defeat Nagle algorithm */
#define TPOPRTIM      5          /* Full Receive Timeout */
#define TPOIPOPT      6          /* IP Option Text */
#define TPOSIOAR      7          /* Add Route */
#define TPOSIODR      8          /* Delete Route */
#define TPOSIFCF      9          /* Interface Config. */
#define TPOSIFLG      10         /* Interface Flags */
#define TPOSIFMT      11         /* Interface MTU */
#define TPOSIFME      12         /* Interface Metric */
#define TPOSIFNM      13         /* Interface Network Mask */
#define TPOSIFBA      14         /* Ifc. Broadcast Address */
#define TPOSIFAD      15         /* Interface Address */
#define TPOSIFEP      16         /* Ifc. Ethernet Address */
#define TPOSIFNO      17         /* Number of Interfaces */
#define TPOSIFDA      18         /* Ifc. Destination Addr. */
#define TPOIPTTL      19         /* IP Time To Live */
#define TPOIPTOS      20         /* IP Type Of Service */
/*
 *
 */
#define TPOLEN      sizeof(struct tpo)

```

tsw

tsw (Transport Endpoint State Word):

C Language Definition

```

struct tsw
{
    unsigned char    tswflags;
    unsigned char    tswpfunc;
    unsigned short   tswstate;
};
/*
 * tswflags defines
 */
#define TSWFCHNG     0x80    /* state is changing */
#define TSWFACPT     0x40    /* accepting to this endpoint */
/*
 * tswpfunc defines (pending functions)
 */
#define TSWPFCLS     0x80    /* TCLOSE */
#define TSWPFDIS     0x40    /* TDISCONN, TCLEAR, TRETRACT */
#define TSWPFREL     0x20    /* TRELEASE */
#define TSWPFACK     0x10    /* TRELACK */
#define TSWPFCON     0x08    /* connection establishment */
#define TSWPFLCL     0x04    /* local endpoint management */
#define TSWPFOPN     0x01    /* TOPEN */
#define TSWPFRCV     0x00    /* TRECVCV */
#define TSWPFSND     0x00    /* TSEND */
#define TSWPFDGM     0x00    /* TSENDTO, TRECVCVFRM, TRECVCVERR */
/*
 * defines for tswstate
 */
#define TSCLOSED     0        /* closed non-existent */
#define TSOPENED     1        /* opened but not bound */
#define TSDSABLD     2        /* bound and disabled */
#define TSENABLD     3        /* bound and enabled */
#define TSINCONN     4        /* connect indication pending */
#define TSOUCONN     5        /* connection in progress */
#define TSCONNCT     6        /* connected or associated */
#define TSINRLSE     7        /* release indication pending */
#define TSOURLSE     8        /* release in progress */
#define TSMAX        TSOURLSE /* max value for tswstate */
/*
 * length of the TSW
 */
#define TSWLEN       sizeof ( struct tsw )

```

tub

tub (Transport Endpoint User Block):

C Language Definition

```
#define TUBUIDLEN    9
#define TUBGRPLEN   9
#define TUBPWDLEN   9
struct tubuids
{
    unsigned char  tubuidl;          /* len of user id */
    char           tubuidc [ TUBUIDLEN-1 ]; /* user id string */
};
struct tubgrps
{
    unsigned char  tubgrpl;          /* len of group id */
    char           tubgrpc [ TUBGRPLEN-1 ]; /* group id string*/
};
struct tubpwds
{
    unsigned char  tubpwdl;          /* len of password */
    char           tubpwdc [ TUBPWDLEN-1 ]; /* user password */
};
/*
 * definition of the TUB
 */
#if 0
#define TUBSTRUCT    1          /* allow less convenient tub */
#endif
#ifndef TUBSTRUCT
struct tub
{
    union
    {
        unsigned char  tubuid [ TUBUIDLEN ];
        struct tubuids tubuids;
    } tubuidu;
    union
    {
        unsigned char  tubgrp [ TUBGRPLEN ];
        struct tubgrps tubgrps;
    } tubgrpu;
    union
    {
        unsigned char  tubpwd [ TUBPWDLEN ];
        struct tubpwds tubpwds;
    } tubpwdu;
};
#else
/*
 * length of the TUB
 */
#define TUBLEN        sizeof ( struct tub )
```

txl

txl (Exit List Structure): C Language Definition

```

/*
 * Common header for txls used with AOPEN and TOPEN
 */
struct txlh
{
    int      txllenxl;          /* length of exit list */
    void     ( *txlconn )();    /* connect indication */
    void     ( *txlconf )();    /* confirm indication */
    void     ( *txldata )();    /* data indication */
    void     ( *txlxdata )();   /* expedited data indication */
    void     ( *txldgerr )();    /* datagram error indication */
    void     ( *txldisc )();    /* disconnect indication */
    void     ( *txlrelease )(); /* orderly release indication */
    void     ( *txlresvd1 )();   /* reserved for future use */
    void     ( *txlresvd2 )();   /* reserved for future use */
    void     ( *txlresvd3 )();   /* reserved for future use */
    void     ( *txltpend )();    /* transport provider end */
    void     ( *txlresvd4 )();   /* reserved for future use */
    void     ( *txlresvd5 )();   /* reserved for future use */
    void     ( *txlresvd6 )();   /* reserved for future use */
};
/*
 * length of TXLH
 */
#define TXLHLEN    sizeof ( struct txlh )
/*
 * definition of the exit list structure used by the TEXTST macro
 */
struct txl
{
    struct txlh      txlh;      /* protocol exit list */
    struct
    {
        int         ( *txlsynad )();    /* physical errors */
        int         ( *txllderad )();   /* logic errors */
    } txlerror;
    void     ( *txlapend )();          /* API subsys end */
    void     ( *txlrsvd7 )();          /* reserved for future */
    void     ( *txlrsvd8 )();          /* reserved for future */
};
/*
 * length of the TXL
 */
#define TXLLEN    sizeof ( struct txl )

```

txp

txp (Transport Exit Parameter List):

C Language Definition

```

struct txp
{
    unsigned short    txptype;        /* exit type */
    unsigned short    txprsvd;       /* resrvd future use */
    union
    {
        unsigned long    txpapcb;    /* APCB pointer */
        unsigned long    txpepid;    /* end point id */
        unsigned long    txptcep;    /* TCEP address */
    } txpep;
    void              (*txpexit)();  /* exit routine entry point */
    union
    {
        unsigned long    txpreasn;   /* TPEND reason code */
        unsigned long    txpevent;   /* protocl event code */
        struct tpl        *txptpl;   /* TPL pointer */
    } txpparm;
    unsigned long      txpacntx;     /* application context */
    unsigned long      txpucntx;     /* user context */
    unsigned long      txpecntx;     /* environment context */
};
/*
 * exit types
 */
#define TXTPROT        1            /* protocol event exit */
#define TXPTCmpl       2            /* endpoint completion exit */
#define TXTPEND        3            /* TP end exit */
#define TXPTSYNc       4            /* synchronous error exit */
#define TXPAPEND       5            /* API subsystem end */
/*
 * protocol event code defines
 */
#define TXPECONN       0            /* connect indication */
#define TXPECONF       4            /* confirm indication */
#define TXPEDATA       8            /* normal data received */
#define TXPEXPDT       12           /* expedited data received */
#define TXPEDISC       20           /* disconnect indication */
#define TXPERLSE       24           /* orderly release indication */
/*
 * TPEND reason codes
 */
#define TXPRDRAN       0            /* operator drained subsystem */
#define TXPRSTOP       4            /* operator stopped subsystem */
#define TXPRTERM       8            /* subsystem terminated */
/*
 * length of TXP
 */
#define TXPLEN         sizeof ( struct txp

```

Socket Library Functions

This chapter describes the BSD Socket Library provided with the Unicenter TCPaccess API. This library supports the communication requirements of application programs written in the C language.

This chapter includes the following sections:

- [Introduction to the Socket Library](#) – Describes the basic terminology and purpose of the socket Application Program Interface (API).
- [Overview of BSD Sockets](#) – Describes communication domains and socket types, creating sockets and binding names, accepting and initiating connections, sending and receiving data, using file I/O functions, shutting down connections, socket and protocol options, non-blocking options, and differences between BSD UNIX and Unicenter TCPaccess sockets.
- [UNIX File I/O Functions](#) – Describes how to use UNIX system calls to read and write sockets as they can read and write disk files.
- [Socket Library Functions](#) – Provides detailed coding information for the API socket functions. Also included in this section are the UNIX file I/O functions that are supported by Unicenter TCPaccess.

The Socket Library

The Unicenter TCPaccess API provides a socket library to support the communication requirements of application programs written in the C language. Generally, these applications are developed in a UNIX environment and ported to operate in an MVS environment. Providing a socket library greatly reduces the necessary changes that must be made to the application program.

Sockets

The term sockets refers to the Application Program Interface (API) implemented for the Berkeley Software Distribution (BSD) of UNIX. The socket interface is used by processes executing under control of the UNIX operating system to access network services in a generalized manner. Sockets support process-to-process communications using a variety of transport mechanisms, including UNIX pipes and Internet and XNS protocols. Socket support is being extended to include ISO protocols.

The socket interface was developed by the University of California at Berkeley and was included with releases 4.1, 4.2, and 4.3 of the BSD UNIX system. Because BSD UNIX has been ported to run on many machine architectures ranging from desktop workstations to large mainframes, many applications use BSD sockets to interface to the communications network. With a variety of C compilers now available for IBM systems, more and more of these applications are being ported to run under the MVS operating system. A library of BSD socket functions can simplify the porting effort by providing the communication services required by these applications, and much of the original C code can be retained.

This chapter describes the BSD Socket Library as provided with the API. The description includes a brief overview of BSD sockets, an explanation of some of the differences between the two implementations, and descriptions of the socket functions included in the library. Familiarity with BSD sockets is assumed. The intent of this chapter is to highlight any differences between the Unicenter TCPaccess API and BSD implementation of sockets and *not* to provide detailed instruction on the use of socket functions.

BSD Sockets

The brief overview of BSD sockets given in this section is not intended to substitute for existing documentation on this subject. In particular, any user of the API socket library is encouraged to obtain and read these documents for a more thorough discussion of sockets and how they are used in the UNIX environment:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*
- *An Advanced 4.3BSD Interprocess Communication Tutorial*

These documents are provided with the 4.3 BSD release of UNIX and are available from:

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Communication Domains and Socket Types

A socket is the UNIX abstraction of a communications endpoint. As such, it must exist within a communications domain identified when the socket is created. A communications domain is represented by the standard set of services provided to the communications endpoints, standardized rules of addressing and protocols used to communicate between endpoints, and the physical communications media. A constant defined in the include file `<socket.h>` is used to identify the communications domain. The communications domain of interest to the API implementation of sockets is the `AF_INET` Internet domain.

Support for ISO-based domains will be added in future releases of the Unicenter TCPaccess API as BSD sockets are extended to accommodate them.

A socket is associated with an abstract type that describes the semantics of communications using the socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the socket type.

Constants

The set of socket types are indicated by these constants defined in <socket.h>:

Socket Type	Description
SOCK_DGRAM	Models the semantics of unreliable datagrams in connectionless-mode communication. Messages may be lost or duplicated and may arrive out of order. A socket of type SOCK_DGRAM requires no connection setup before communication can begin and can communicate with multiple peers.
SOCK_RDM	Models the semantics of reliable datagrams in connectionless-mode communication. Messages arrive unduplicated and in order, and the sender is notified if messages are lost. A socket of type SOCK_RDM requires no connection setup before communication can begin, and can communicate with multiple peers.
SOCK_STREAM	Models connection-based virtual circuits without record boundaries and provides reliable two-way transfer of ordered byte streams without duplication of data and without preservation of boundaries. A socket of type SOCK_STREAM communicates with a single peer and requires connection setup before communication can begin.
SOCK_SEQPACKET	Models connection-based virtual circuits with record boundaries and provides reliable two-way transfer of ordered byte streams without duplication of data, while preserving boundaries within the data stream. A socket of type SOCK_SEQPACKET communicates with a single peer and requires connection setup before communication may begin.

Socket Type	Description
SOCK_RAW	Models connectionless-mode and is normally used with the <code>sendto()</code> and <code>recvfrom()</code> socket calls. The <code>connect()</code> call may be used to fix the destination for future datagrams. If the <code>connect()</code> call is used, the <code>read()</code> or <code>recv()</code> and <code>write()</code> or <code>send()</code> calls may be used. The application must provide a complete IP header when sending. Otherwise, <code>IPPROTO_RAW</code> will be set in outgoing datagrams and used to filter incoming datagrams and an IP header will be generated and prepended to each outgoing datagram. In either case, received datagrams are returned with the IP header and options intact.
SOCK_ASSOC	Models the semantics of unreliable datagrams in connectionless-mode communication with associations. Messages may be lost or duplicated and may arrive out of order. Formerly <code>SOCK_DGRAM</code> in previous versions of Unicenter TCPaccess sockets.
SOCK_CONNLESS	No longer used, but is included for backwards compatibility. Note: <code>SOCK_DGRAM</code> replaces <code>SOCK_CONNLESS</code> . Sockets of type <code>SOCK_DGRAM</code> and <code>SOCK_RDM</code> are generally more efficient and are most useful in transaction-based applications where repetitive connection setup and breakdown should be avoided. Sockets of type <code>SOCK_STREAM</code> and <code>SOCK_SEQPACKET</code> support an out-of-band transmission facility that can be used to send expedited data.

Note: `SOCK_RDM` and `SOCK_SEQPACKET` are not supported by Unicenter TCPaccess sockets.

Each socket can have a specific protocol associated with it. This protocol is used within the domain to provide the semantics required by the socket type. Not all socket types are supported by each domain; support depends on the existence and the implementation of a suitable protocol within the domain. For example, within the Internet domain, the `SOCK_DGRAM` type may be implemented with Transmission Control Protocol (TCP). No standard protocols exist in the Internet domain to implement the `SOCK_RDM` and `SOCK_SEQPACKET` socket types.

Creating Sockets and Binding Names

A socket is created with the `socket` function:

```
s = socket ( domain, type, protocol );  
int s, domain, type, protocol;
```

Communications Domain and Socket Type

The communications domain and socket type are specified using the constants defined in `<socket.h>`. The protocol may be specified as zero, indicating any suitable protocol for the given domain and type.

One of several possible protocols can be specified by indicating its protocol number obtained by this library function:

```
getprotobyname().
```

Socket Descriptor

A socket descriptor, `s`, is returned by the `socket` function and should be used in all subsequent functions that reference the socket. The socket is initially unconnected, and if it was created with a connectionless type, datagrams can be sent and received immediately without establishing a connection. Otherwise, the socket is connection-oriented and must become connected before sending and receiving data.

An unconnected socket becomes connected in one of two ways:

- By actively connecting to another socket
- By becoming bound to a name in the communications domain and accepting a connection from another socket

Communications Domain Name

A name is currently limited to 16 bytes and has this structure:

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

The address family, `sa_family`, identifies the domain within which the name exists and is selected from a set of constants similar to those used to specify the domain when a socket is created.

The format and content of a name is domain-dependent. Generally the name consists of network, host, and protocol addresses. For the Internet domain, `AF_INET`, a name has this structure (defined in `<inet.h>`):

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

Note: For 3.1 and higher, `AF_INET` has been redefined to be 2 so as to be compatible with most UNIX programs. For backwards compatibility, sockets will accept either 1 (the previous value) or 2 for `AF_INET`.

For historical reasons, the Internet address `sin_addr` is defined as the structure `in_addr`:

```
struct in_addr
{
    u_long s_addr;
};
```

To accept connections or receive datagrams, a socket must be bound to a name (or address) within the communications domain. To initiate a connection or send datagrams, the application program may bind the name or let the system bind a default name.

A name is bound using the `bind` function:

```
bind ( s, name, namelen );

int s;
struct sockaddr *name;
int namelen;
```

Whether the name was specified by the application program or assigned by the system (that is, the API), any name bound to a socket can be retrieved with the `getsockname` function:

```
getsockname ( s, name, namelen );  
  
int s;  
struct sockaddr *name;  
int namelen;
```

The name of the connected peer can be retrieved with the `getpeername` function:

```
getpeername ( s, name, namelen );  
  
int s;  
struct sockaddr *name;  
int namelen;
```

Accepting and Initiating Connections

After a name is bound to a connection-oriented socket, a connection must be established before the application program can send or receive data. The method used to establish a connection depends on the operating mode of the program.

Operating in Server Mode

Programs operating in server mode generally are passive and listen for connection requests using the `listen` function:

```
listen ( s, backlog );  
  
int s, backlog;
```

The `backlog` argument specifies the maximum number of connection requests that can be queued simultaneously awaiting acceptance.

A connection request is accepted by executing an `accept` function:

```
ns = accept ( s, name, namelen );  
  
int ns, s;  
struct sockaddr *name;  
int namelen;
```

The accepted connection is established to a new socket that has the same characteristics (that is, domain, type, and protocol) as the listening socket. The socket descriptor of the new socket, `ns`, is used in subsequent `send` and `recv` functions to exchange data with the connected peer, and the old socket descriptor, `s`, can be reused to listen for another connection request.

Operating in Client Mode

Programs operating in client mode generally are active and initiate a connection to a peer process by specifying the name of the peer with a connect function:

```
connect ( s, name, namelen );  
  
int s;  
struct sockaddr *name;  
int namelen;
```

Once Connection is Established

The server and client modes of operation affect only how connections are established. Once a socket is connected, send and receive operations are executed without regard to how the connection was established. The mode of connection used by each peer process is agreed to in advance.

Although sockets of the connectionless datagram type do not establish real connections, the connect function can be used with such sockets to create an association with a particular peer. The name provided with the connect function is recorded for use in future send functions, which then need not supply the destination name. Only datagrams arriving at the socket from the associated peer are queued for receiving with subsequent recv functions.

Sending and Receiving Data

Messages can be sent and received from both connected and unconnected sockets.

Sending Messages from Unconnected Sockets

Messages can be sent from unconnected sockets using the sendto function:

```
cc = sendto ( s, buf, len, flags, to, tolen );  
  
int cc, s;  
char *buf;  
int len, flags;  
struct sockaddr *to;  
int tolen;
```

When sending on unconnected sockets, to and tolen indicate the destination of the message. The number of bytes sent is returned.

Sending Messages from Connected Sockets

Messages can be sent from connected sockets using the `send` function:

```
cc = send ( s, buf, len, flags );  
  
int cc, s;  
char *buf;  
int len, flags;
```

The message to be sent is identified by the `buf` and `len` arguments, and the `flags` argument is used to indicate normal or out-of-band data.

Receiving Messages from Unconnected Sockets

Messages can be received from unconnected sockets using the `recvfrom` function:

```
cc = recvfrom ( s, buf, len, flags, from, fromlen );  
  
int cc, s;  
char *buf;  
int len, flags;  
struct sockaddr *from;  
int fromlen;
```

The receive buffer is identified by the `buf` and `len` arguments, and `flags` indicate whether the buffer is to be used for receiving normal or out-of-band data. When receiving on unconnected sockets, `from` and `fromlen` identify a buffer for returning the source of the message. The function return value is the length of the message received.

Receiving Messages from Connected Sockets

Messages can be received from connected sockets using the `recv` function:

```
cc = recv ( s, buf, len, flags );  
  
int cc, s;  
char *buf;  
int len, flags;
```

The receive buffer is identified by the `buf` and `len` arguments, and `flags` indicate whether the buffer is to be used for receiving normal or out-of-band data.

Sending and Receiving Messages from Noncontiguous Buffers

The functions previously described provide for the sending and receiving of messages whose contents are stored in contiguous memory buffers. You can also send and receive messages gathered from or scattered into noncontiguous buffers using the `sendmsg` and `recvmsg` functions:

```
cc = sendmsg ( s, msg, flags );
int cc, s;
struct msghdr *msg;
int flags;
cc = recvmsg ( s, msg, flags );
int cc, s;
struct msghdr *msg;
int flags;
```

The structure `msghdr` is used to pass several parameters to `sendmsg` and `recvmsg` that reduce the number of direct function arguments:

```
struct msghdr
{
    char          *msg_name;
    int           msg_namelen;
    struct iovec  *msg_iov;
    int          msg_iovlen;
    char          *msg_accrights;
    int          msg_accrightslen;
};
```

The list of noncontiguous buffer segments is defined by an array of `iovec` structures:

```
struct iovec
{
    char *iov_base;
    int  iov_len;
};
```

The `sendmsg` and `recvmsg` functions can be used with connected or unconnected sockets. If the socket is unconnected, `msg_name` and `msg_namelen` specify the destination or source of the message. Otherwise, these arguments should be null. `msg_iov` specifies an array of noncontiguous buffer segments, and `msg_iovlen` is the length of the array. The `msg_accrights` and `msg_accrightslen` arguments are used to specify access rights for sockets in the UNIX domain (`AF_UNIX`) and should be null for the Internet (`AF_INET`) domain.

Using File I/O Functions

Standard UNIX file I/O functions can be used to read and write sockets as if they were UNIX files. The socket descriptor is used in place of a file descriptor.

The read and write Functions

The read and write functions are used to receive and send contiguous data:

```
cc = read ( s, buf, len );  
  
int cc, s;  
char *buf;  
int len;  
cc = write ( s, buf, len );  
int cc, s;  
char *buf;  
int len;
```

The readv and writev Functions

The readv and writev functions are used to receive and send noncontiguous data:

```
cc = readv ( s, iov, iovcnt );  
  
int cc, s;  
struct iovec *iov;  
int iovcnt;  
cc = writev ( s, iov, iovcnt );  
int cc, s;  
struct iovec *iov;  
int iovcnt;
```

The array specified by `iov` contains `iovcnt` structures, each of which defines the beginning address and length of a buffer segment:

```
struct iovec  
{  
    char    *iov_base;  
    int     iov_len;  
};
```

By using the `read`, `readv`, `write`, and `writev` functions, the application program can read from and write to sockets, terminals, and files without distinguishing the descriptor type.

Shutting Down Connections

An application program that no longer needs a connected socket can gracefully shut the connection down using the shutdown function:

```
shutdown ( s, direction );  
int s, direction;
```

To discontinue receiving, set direction to zero.

To discontinue sending, set direction to one.

To shut the connection down in both directions, set direction to two.

If the underlying protocol supports unidirectional or bidirectional shutdown, this indication is passed to the peer.

Note: A socket will not terminate until a close() has been issued for the socket.

A connection can also be gracefully closed and the socket eliminated from the system by using the close function:

```
close ( s );  
int s;
```

Socket and Protocol Options

Sockets, and the underlying protocols, can support options. These options control implementation or protocol-specific facilities. The getsockopt and setsockopt functions are used to manipulate these options:

```
getsockopt ( s, level, optname, optval, optlen )  
int s, level, optname;  
char *optval;  
int *optlen;  
setsockopt ( s, level, optname, optval, optlen )  
int s, level, optname;  
char *optval;  
int optlen;
```

The option optname is interpreted at the indicated protocol level for sockets. If a value is specified with optval and optlen, it is interpreted by the software operating at the specified level. The level SOL_SOCKET is reserved to indicate options maintained by the socket facilities. Other level values indicate a particular protocol that is to act on the option request; these values normally are interpreted as a protocol number obtained with getprotobyname.

Non-blocking I/O

Socket functions that cannot complete immediately because of flow control or synchronization requirements block the application program until the operation can be completed. This causes the task that issued the function to be suspended.

If blocking is not desired, the non-blocking mode can be selected for a socket using the `ioctl` function:

```
ioctl ( s, request, argp );  
  
int s;  
unsigned long request;  
char *argp;
```

If request is set to the constant `FIONBIO`, defined in `<socket.h>`, and the boolean argument specified by `argp` is true (that is, non-zero), the socket is enabled for non-blocking I/O. When operating in this mode, functions that would otherwise block are completed with an error code indicating the blocking condition. The application can then reissue the function at a later time when it can complete without blocking.

MVS vs. UNIX

Many subtle differences exist between the Unicenter TCPaccess the BSD UNIX socket implementations. For most users, these differences go unnoticed, but for the more sophisticated application these differences should be noted and accounted for. Another area that could cause the BSD UNIX socket programmer some problems is realizing where sockets end and the UNIX operating system begins. The programmer must understand the functioning of the MVS operating system as opposed to the UNIX environment.

Here are some of the OS environment differences that must be realized before porting a UNIX socket application to the MVS environment:

- UNIX sockets are part of the kernel and run in supervisor mode. Unicenter TCPaccess sockets are simply functions that the user program links to and therefore run in user mode. Unicenter TCPaccess sockets do make calls to the Unicenter TCPaccess subsystem via PC calls that run in a semi-privilege mode. The main item to note is that Unicenter TCPaccess sockets do not have the privilege or protection of BSD sockets because they run in user mode. Thus it is impossible for them to check for access violations when passed pointers to structures and buffers. Some of these types of errors can be caught by the API, while others cause the user program to ABEND. UNIX sockets are able to verify the validity of pointers and return an error (EFAULT) if the pointer would allow access to a privileged or nonexistent area of memory.

- The UNIX operating system lets a process fork a child process that is an exact copy of the parent. This lets a parent open a socket or file and fork a child that then has total access to this open socket or file. The MVS OS does not provide this feature. In MVS a task can ATTACH another task, but the two tasks must then coordinate access to any variables that the two may share. Therefore it is impossible with sockets under MVS to pass a socket from one task to another transparently. For programs that require multiple tasks using multiple endpoints, the designer should consider an architecture where one task is placed in charge of controlling the network activity; some form of inter-task communication could be used where other tasks can inform the task in charge about when to open and close endpoints, the location of data buffers for input and output, and other control information for the endpoints, or using the augmented socket library routines provided to transfer ownership of the socket from one task to another. Sockets cannot be passed across address spaces.
- When a process terminates abnormally or does not close the sockets or files it created during execution, the UNIX OS closes these gracefully on termination of the process. With Unicenter TCPaccess sockets, the termination routine is called via an atexit() entry point, and the open sockets are closed abruptly. This may be viewed by the remote endpoint as an abortive disconnect. To alleviate this problem as much as possible, socket library programmers should explicitly close all sockets their programs create before exiting. If the program terminates abnormally, users should be aware of the consequences.

General Socket Differences

This subsection covers most of the general differences between BSD UNIX sockets and Unicenter TCPaccess sockets. In the later discussion of each function, differences between Unicenter TCPaccess and BSD socket functions are listed.

Supported Sockets

Unicenter TCPaccess sockets are supported only in the Internet domain (AF_INET).

The five types of sockets supported are listed in the following table:

Socket Type	Description
SOCK_STREAM	Use TCP for transport and provide a reliable byte stream service.
SOCK_DGRAM	Use UDP for transport and provide an unreliable message service in connectionless mode only. Formerly SOCK_CONNLESS in previous releases.
SOCK_CONNLESS	Use UDP for transport and provide an unreliable message service in connectionless mode only. No longer actively supported, but provided for backward compatibility - use SOCK_DGRAM
SOCK_ASSOC	Use UDP for transport and provide an unreliable message service with associations only. Formerly SOCK_DGRAM
SOCK_RAW	Use with UDP for transport and provide an unreliable message service.

SOCK_DGRAM provides the same functionality as SOCK_DGRAM sockets in the UNIX world. As an added feature, it is possible for sockets of the type SOCK_ASSOC to simulate a connection-based server. A user can issue a listen() function and when a datagram is received on this endpoint, the socket library simulates a pending connection request that the user can acknowledge with the accept() function call. This facility is not provided with UNIX implementation of sockets.

Note: The usage of `SOCK_DGRAM` and `SOCK_CONNLESS` has changed for release 3.1 and higher of Unicenter TCPaccess and Unicenter TCPaccess sockets. Existing programs using `SOCK_CONNLESS` should be re-compiled and relinked substituting `SOCK_DGRAM`, although for backwards compatibility, `SOCK_CONNLESS` has been mapped to `SOCK_DGRAM`. Socket programs using associations previously used `SOCK_DGRAM`. These programs should be re-compiled and re-linked to use `SOCK_ASSOC`.

Endpoints

Unicenter TCPaccess defines endpoints below 4096 as server endpoints only. On a socket `bind()` the user can request an endpoint below 4096 only. This endpoint can act only as a server. If no endpoint is requested, an endpoint with port number 4096 or above is assigned. This endpoint can act only as a client.

Binding a Name to a Socket

When binding a name to a socket, only a port number is allowed. The Internet address must be set to `INADDR_ANY(0)`. Unicenter TCPaccess assigns the proper address to the endpoint when it can determine which of the possible network interfaces to use.

Urgent Data

The UNIX concept of urgent data – Out-of-Band (OOB) – is not supported in the same fashion in the Unicenter TCPaccess socket implementation. UNIX interprets OOB data to be a single byte of data within the stream. The UNIX socket programmer can specify that this byte of data be read in-band or out-of-band. The Unicenter TCPaccess subsystem does not allow for the urgent data being read out-of-band. Therefore, you cannot specify the type of data to read with any of the receive request functions (for example, `MSG_OOB` is not supported, nor is `SO_OOBINLINE` option). Requests specifying either of these parameters do not cause an error, but do not have any effect on the operation of the socket library. In effect, the Unicenter TCPaccess socket implementation operates as if the `SO_OOBINLINE` socket option has been set.

For SAS/C compiler users, these methods are available to determine if urgent data has been received:

- `SIGURG` signal

This signal is generated by the socket library when urgent data is received.

- `select()` on exceptional conditions

If a `select()` call completes noting read availability and a pending exceptional condition, then urgent data is available to be read.

For IBM compiler users, only the `select()` on exceptional conditions method is available for determining if urgent data has been received.

The `ioctl()` request `SIOCATMARK` can be used to determine specifically whether the socket read pointer points to the last type of urgent data (the read pointer points to the data to be read by the next socket read-oriented operation).

Error Codes

The error codes set in the global variable, `errno`, are all defined with the prefix `E` to indicate “error”. Error codes are mapped to the corresponding error code for the compiler that you are using, either `SAS/C` or `IBM/C370`.

Note: Versions of Unicenter TCPaccess sockets prior to release 3.1 had error codes prefixed with `ES` and added a base error code. The current release of Unicenter TCPaccess uses only `E` as the prefix to be more compatible with UNIX sockets and maps the error code according to the compiler you are using, either `SAS/C` or `IBM/C370`. For backward compatibility, the `ES` prefix will be accepted.

The value for `errno` may now be checked directly. A macro, `GET_ERRNO` had been provided in earlier releases for querying the value of `errno`. This macro is still available

```
if ( ( s = socket ( AF_INET, SOCK_STREAM, 0 ) ) < 0 )
{
    if ( GET_ERRNO == ESFAULT )
    {
        /* do something special for this type of error */
    }
}
```

Extra Functions

These extra functions have been added to the socket library to assist in the development of applications using the socket library:

- `mvselect()`
Provides the same features as `select()`, plus it allows selecting on an optional generic ECB list.
- `closepass()` and `openold()`
When used together, lets a socket be passed to another task.

ANSI-C Compatible Function Prototypes

The socket library functions all have ANSI-compatible function prototypes associated with them. These prototypes provide for better error checking by the C compiler. However, to the pre-ANSI programmer, these may be more of an annoyance than a benefit. If you want to ignore these function prototypes, you can define the term NOSLIBCK at compile time and the ANSI function prototypes are ignored and the more common function definitions are used.

Socket Header Files

Socket Header File	Description
acs.h	This header file is no longer required but is included for backward compatibility. It simply includes socket.h.
icssckt.h	This header file is for use with Unicenter TCPaccess UNIX System Services sockets. See the chapter titled "UNIX System Services MVS Integrated System Sockets" for more information.
inet.h	This header file takes the place of both in.h and inet.h (for those familiar with the UNIX header file organization). It defines values common to the Internet. This header file takes the place of both <netinet/in.h> and <inet.h> on UNIX.
ip.h	This header file defines values used by the Inter-network Protocol (IP) and details the format of an IP header and options associated with IP. The current implementation of the socket library does not let the socket library user access the IP layer, and therefore, this header file is of little use for the time being. Later versions of the socket library will provide the ability to set IP options and at that time this library will be required. This file is the same as <netinet/ip.h> on UNIX.
netdb.h	This header file defines the structures used by the "get" services. It also provides the function prototypes for this family of functions. This file has the same functionality as <netdb.h> on UNIX.
errno.h	This header file defines the errors that can be returned by the socket library when requests are made to it. The value of errno can be determined directly, or, as in previous releases, the macro GET_ERRNO provides for the proper access to the errno variable of the socket library. This file has the same functionality as <errno.h> on UNIX.
sockcfg.h	This header file describes the socket configuration structure. This file has no equivalent on UNIX.
socket.h	This header file defines most of the variables and structures required to interface properly to the socket library. It also provides the function prototypes in ANSI form for those functions that are truly socket functions. This file takes the place of <sys/types.h>, <sys/socket.h>, <sys/time.h>, <sys/param.h>, <sys/stat.h>, <sys/ioctl.h>, and <sys/fcntl.h> on UNIX.

Socket Header File	Description
sockvar.h	This header file is needed when compiling the socket configuration file (sockcfg.c) and also is used internally by the socket library. It is not needed to interface to the socket library. This file has the same functionality as <sys/socketvar.h> on UNIX.
tcp.h	This header file describes those options that may be set for a socket of type SOCK_STREAM. This file has the same functionality as <netinet/tcp.h> on UNIX.
uio.h	This header file describes the structures necessary to use vectored buffering of the socket library. Due to ANSI checking, this header file must be included in any header file that includes socket.h. This file has the same functionality as <sys/uio.h> on UNIX.
user.h	This header file is needed when compiling the socket configuration file (sockcfg.c) and is used internally by the socket library. It is not needed to interface to the socket library. There is no equivalent for this file on UNIX.

Options

This implementation of sockets does not support all of the option facilities of the BSD UNIX version. No options are supported at the IP or UDP levels. In addition, no options can be set at the interface or driver levels. At the socket level (SOL_SOCKET), some additional options have been added. For further explanation of those options supported by this implementation, see [getsockopt\(\)](#) and [setsockopt\(\)](#).

Error Codes

Various error codes that are stored in the external variable `errno` have been added. Those that are particular to a function call are detailed in the section describing the function. These error codes can occur with any function call:

- **ECONFIG**

This error can occur when the application first opens a socket. It indicates that an error was detected with the socket configuration or when initializing a user session with the API. This type of error occurs only on a call to the `socket()` function. To further isolate this type of error, users should configure their socket configurations to allow extended error messages (That is, set `EXTERRNOMSGS` and `CONFIGDEBUG` on in `sockcfg.flags`) and configuration debug. Once an error occurs, user programs should issue a call to `perror()` to view the error message generated. Details and troubleshooting action for this type of problem are covered in the *Messages and Codes*.

- ESYS

This error can occur when the underlying API malfunctions. If, during normal operation, the API returns an error code that should not occur during the function execution, this error is returned. The user should ensure the proper operation of the API or Unicenter TCPAccess before trying to isolate this problem further.

- ETPEND

This error code is set when the API is stopped or terminates abnormally. The user program should close all endpoints at this time. At some interval the user program can attempt to reopen sockets to see if the API has been restarted. When ETPEND is no longer returned, the API has restarted and the user program can continue to use the network facilities.

UNIX File I/O Functions

BSD UNIX integrates socket and file I/O facilities in such a way that UNIX system calls that normally are used to read and write disk files can also be used to read and write sockets. This is particularly convenient when a process inherits a socket or file descriptor from another process. The inheriting process can process the input or output stream without knowing whether it is associated with a socket or file.

Example

A filtering process that receives its input from STDIN and writes its output to STDOUT need not distinguish socket descriptors from file descriptors.

Using UNIX Routines in MVS

In a standard UNIX environment, if a file I/O function is issued using a socket descriptor in place of the normal file descriptor, the file I/O routine can route the request to the appropriate socket I/O routine because both execute in the UNIX kernel. However, in an MVS environment where UNIX file I/O routines are being simulated, restrictions might exist.

UNIX file I/O operations are simulated by library routines that translate I/O requests into operations compatible with MVS access methods. Generally, these library routines are provided by the vendor of the C compiler used to compile the application program. Therefore, the socket library functions provided with the API must be integrated with these routines if the application program intends to use UNIX file I/O functions to manipulate sockets and files at the same time.

Integrating API Socket Functions with UNIX File I/O

One of these techniques can be used to integrate the Unicenter TCPaccess socket functions with UNIX file I/O:

- The file I/O function provided with the C compiler runtime library can be modified to call the appropriate socket function if the descriptor is associated with a socket. This presumes that the compiler vendor provides source code for the runtime library.
- If source code is not available, the technique is to front-end the UNIX file I/O function with a similarly-named function and to rename the original member in the library. Intercepted requests can then be routed to the appropriate function.

Descriptions are provided for those UNIX file I/O functions that also operate on sockets and are supported by the Unicenter TCPaccess API. Since source code is furnished for these library functions, they can serve as the basis for front-ending existing functions that simulate UNIX file I/O. The descriptions provided apply only when the associated function is used to operate on a socket.

Socket Library Functions

This section provides detailed coding information for the Unicenter TCPaccess socket functions.

The socket library functions are presented in alphabetical order. Each page that pertains to a particular function has the name of the function in the upper outside corner.

Because of the different operating system environment within which the API sockets must operate, a few additional functions have been included in the socket library. These functions simplify integration of the socket library into the IBM MVS environment.

Note: These functions are `closepass()`, `mvselect()`, and `openold()`, and are detailed in the socket functions section.

Component Functional Description

Following a brief introductory statement summarizing its use, each function is described using the documentation style of UNIX. This table lists the basic components of each function description:

Synopsis

A synopsis of the function is given in C language format. The function prototype statement is listed showing all function arguments, followed by declaration statements defining the format of each argument. If the function returns a value other than the normal success/failure indication, the function prototype is shown as an assignment statement.

Description

A description of the function is given, including any special rules for specifying arguments, alternative uses of the function, and any results returned.

Return Value

The function's return value, if any, is defined.

Error Codes

Error codes that are returned when the function completes abnormally are defined. Generally, these are error codes defined as constants in `<errno.h>` and are returned in the global integer `errno`.

Implementation Notes

Any difference between the Unicenter TCPaccess API and BSD implementation of a function is noted here. Side effects that are a consequence of the operating system environment are also noted. If no implementation notes are listed, the Unicenter TCPaccess API and BSD implementations are functionally equivalent.

See Also

References to related functions are given.

accept()

Accept a connection on a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
int accept ( s , name, namelen )
int s;
struct sockaddr *name;
int *namelen;
```

Description

Parameters:

- s** A socket created with `socket()`, bound to a name with `bind()`, and listening for connections after a `listen()`.
- name** A result parameter that is filled in with the name of the connecting entity, as known to the communications layer. The exact format of the name parameter is determined by the domain in which the communication is occurring.
- namelen** A value-result parameter and should initially contain the amount of space pointed to by `name`; on return it contains the actual length (in bytes) of the name returned. This function is used with connection-based or association-based socket types, currently with `SOCK_STREAM` and `SOCK_ASSOC`, respectively.

The `accept()` function is used to accept a pending connection request queued for a socket that is listening for connections. A new socket is created for the connection, and the old socket continues to be used to queue connection requests.

The `accept()` function extracts the first connection on the queue of pending connections, creates a new socket with the same properties of `s`, and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `accept()` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `accept()` returns an error. The accepted socket, `ns`, cannot be used to accept more connections. The original socket `s` remains open.

It is possible to select() a socket for the purposes of doing an `accept()` by selecting it for read.

Return Value

If the `accept()` function succeeds – returns a non-negative integer that is a descriptor for the accepted socket. Otherwise, the value -1 is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes	<p>The <code>accept()</code> function returns the following error codes (from <code><errno.h></code>) in the global integer <code>errno</code>:</p> <table><tr><td><code>EBADF</code></td><td>The descriptor is invalid.</td></tr><tr><td><code>ECONNABORTED</code></td><td>The incoming connection request was aborted by the remote endpoint.</td></tr><tr><td><code>ECONNREFUSED</code></td><td>The remote endpoint refused to complete the connection sequence.</td></tr><tr><td><code>ECONNRESET</code></td><td>The remote endpoint reset the connection request.</td></tr><tr><td><code>EDESTUNREACH</code></td><td>Remote destination is now unreachable.</td></tr><tr><td><code>EFAULT</code></td><td>The name pointer, name, or the name length pointer, <code>namelen</code>, points to inaccessible storage.</td></tr><tr><td><code>EHOSTUNREACH</code></td><td>Remote host is now unreachable.</td></tr><tr><td><code>EINVAL</code></td><td>The socket is not listening for connections.</td></tr><tr><td><code>EMFILE</code></td><td>The socket descriptor table is full. There is no room to save the new socket descriptor that <code>accept()</code> normally returns.</td></tr><tr><td><code>ENFILE</code></td><td>New socket cannot be opened due to API resource shortage or user endpoint allocation limits.</td></tr><tr><td><code>ENETDOWN</code></td><td>Local network interface is down.</td></tr><tr><td><code>ENOMEM</code></td><td>No memory is available to allocate space for the new socket.</td></tr><tr><td><code>EOPNOTSUPP</code></td><td>The referenced socket is not of type <code>SOCK_STREAM</code> or <code>SOCK_ASSOC</code>.</td></tr><tr><td><code>ETIMEDOUT</code></td><td>The connection request by the remote endpoint timed out.</td></tr><tr><td><code>ETIMEDOUT</code></td><td>Remote endpoint timed out connection.</td></tr><tr><td><code>EWOULDBLOCK</code></td><td>The socket is marked non-blocking and no connections are present to be accepted.</td></tr></table>	<code>EBADF</code>	The descriptor is invalid.	<code>ECONNABORTED</code>	The incoming connection request was aborted by the remote endpoint.	<code>ECONNREFUSED</code>	The remote endpoint refused to complete the connection sequence.	<code>ECONNRESET</code>	The remote endpoint reset the connection request.	<code>EDESTUNREACH</code>	Remote destination is now unreachable.	<code>EFAULT</code>	The name pointer, name, or the name length pointer, <code>namelen</code> , points to inaccessible storage.	<code>EHOSTUNREACH</code>	Remote host is now unreachable.	<code>EINVAL</code>	The socket is not listening for connections.	<code>EMFILE</code>	The socket descriptor table is full. There is no room to save the new socket descriptor that <code>accept()</code> normally returns.	<code>ENFILE</code>	New socket cannot be opened due to API resource shortage or user endpoint allocation limits.	<code>ENETDOWN</code>	Local network interface is down.	<code>ENOMEM</code>	No memory is available to allocate space for the new socket.	<code>EOPNOTSUPP</code>	The referenced socket is not of type <code>SOCK_STREAM</code> or <code>SOCK_ASSOC</code> .	<code>ETIMEDOUT</code>	The connection request by the remote endpoint timed out.	<code>ETIMEDOUT</code>	Remote endpoint timed out connection.	<code>EWOULDBLOCK</code>	The socket is marked non-blocking and no connections are present to be accepted.
<code>EBADF</code>	The descriptor is invalid.																																
<code>ECONNABORTED</code>	The incoming connection request was aborted by the remote endpoint.																																
<code>ECONNREFUSED</code>	The remote endpoint refused to complete the connection sequence.																																
<code>ECONNRESET</code>	The remote endpoint reset the connection request.																																
<code>EDESTUNREACH</code>	Remote destination is now unreachable.																																
<code>EFAULT</code>	The name pointer, name, or the name length pointer, <code>namelen</code> , points to inaccessible storage.																																
<code>EHOSTUNREACH</code>	Remote host is now unreachable.																																
<code>EINVAL</code>	The socket is not listening for connections.																																
<code>EMFILE</code>	The socket descriptor table is full. There is no room to save the new socket descriptor that <code>accept()</code> normally returns.																																
<code>ENFILE</code>	New socket cannot be opened due to API resource shortage or user endpoint allocation limits.																																
<code>ENETDOWN</code>	Local network interface is down.																																
<code>ENOMEM</code>	No memory is available to allocate space for the new socket.																																
<code>EOPNOTSUPP</code>	The referenced socket is not of type <code>SOCK_STREAM</code> or <code>SOCK_ASSOC</code> .																																
<code>ETIMEDOUT</code>	The connection request by the remote endpoint timed out.																																
<code>ETIMEDOUT</code>	Remote endpoint timed out connection.																																
<code>EWOULDBLOCK</code>	The socket is marked non-blocking and no connections are present to be accepted.																																
Implementation Notes	Unlike UNIX sockets, this implementation does let <code>SOCK_ASSOC</code> (UDP sockets using association-oriented operation) listen for and accept connections.																																
See Also	<code>bind()</code> , <code>connect()</code> , <code>listen()</code> , <code>select()</code> , <code>socket()</code> , <code>closepass()</code> , <code>openold()</code>																																

bind()

Bind a name to a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
int bind ( s, name, namelen )
int s;
struct sockaddr *name;
int namelen;
```

Description

Assigns a name to an unnamed socket that represents the address of the local communications endpoint. For sockets of type SOCK_STREAM, the name of the remote endpoint is assigned when a connect() or accept() function is executed.

When a socket is created with socket(), it exists in a name space (address family), but has no name assigned. bind() requests that name be assigned to the socket. The rules used in name binding vary between communication domains.

Return Value

If bind() is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The bind() function returns these error codes (from <errno.h>) in the global integer errno:

EACCES	The requested address is protected, and the current user has inadequate permission to access it.
EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available. This can occur if the address portion of the name is not equal to INADDR_ANY or the port is greater than or equal to IPPORT_RESERVED.
EAFNOSUPPORT	The address family requested in the name does not equal AF_INET.
EBADF	The argument s is not a valid descriptor.
EFAULT	The pointer, name, points to inaccessible memory.
EINVAL	The socket is already bound to an address.
EINVAL	The length of the name, namelen, does not equal the size of a sockaddr_in structure.

Implementation Notes	Currently, only names from the Internet domain, AF_INET, may be bound to a socket. The address portion of the name must equal INADDR_ANY, else EINVAL is returned. Server and client ports are reserved differently with this implementation than with UNIX. Ports 0 to 4095 are reserved for server ports, and 4096 and up are for client ports. The user should only assign a port when acting as a server and should specify a port of zero when doing a bind() for a client port.
See Also	connect(), listen(), socket(), getsockname()

close()

	Delete a socket.
Synopsis	<pre>#include <socket.h> #include <uio.h> int close (s) int s;</pre>
Description	<p>Deletes a socket descriptor created by the socket() function. On completion, the endpoint no longer exists in the communications domain.</p> <p>close() deletes the socket descriptor, s, from the internal descriptor table maintained for the application program and terminates the existence of the communications endpoint. If the socket was connected, the connection is terminated. The connection is released as much as possible, in an orderly fashion. Data that has yet to be delivered to the remote endpoint remains queued, because the endpoint tries to deliver it before a timeout causes the local endpoint to be destroyed and removed from the system.</p> <p>Sockets use MVS STIMER services. Socket applications may use up to fifteen STIMER calls per task control block (TCB). STIMER CANCEL ID=ALL must not be used by socket applications.</p>

Return Value	<p>If <code>close()</code> is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer <code>errno</code> indicates the nature of the error.</p> <table><tr><td>EBADF</td><td>The <code>s</code> argument is not an active descriptor.</td></tr><tr><td>ECONNABORTED</td><td>The connection was aborted by the remote endpoint.</td></tr><tr><td>ECONNREFUSED</td><td>The remote endpoint refused to continue the connection.</td></tr><tr><td>ECONNRESET</td><td>The remote endpoint reset the connection request.</td></tr><tr><td>EDESTUNREACH</td><td>Remote destination is now unreachable.</td></tr><tr><td>EHOSTUNREACH</td><td>Remote host is now unreachable.</td></tr><tr><td>ENETDOWN</td><td>Local network interface is down.</td></tr><tr><td>ETIMEDOUT</td><td>The connection timed out.</td></tr></table>	EBADF	The <code>s</code> argument is not an active descriptor.	ECONNABORTED	The connection was aborted by the remote endpoint.	ECONNREFUSED	The remote endpoint refused to continue the connection.	ECONNRESET	The remote endpoint reset the connection request.	EDESTUNREACH	Remote destination is now unreachable.	EHOSTUNREACH	Remote host is now unreachable.	ENETDOWN	Local network interface is down.	ETIMEDOUT	The connection timed out.
EBADF	The <code>s</code> argument is not an active descriptor.																
ECONNABORTED	The connection was aborted by the remote endpoint.																
ECONNREFUSED	The remote endpoint refused to continue the connection.																
ECONNRESET	The remote endpoint reset the connection request.																
EDESTUNREACH	Remote destination is now unreachable.																
EHOSTUNREACH	Remote host is now unreachable.																
ENETDOWN	Local network interface is down.																
ETIMEDOUT	The connection timed out.																
See Also	<code>accept()</code> , <code>bind()</code> , <code>connect()</code> , <code>shutdown()</code> , <code>socket()</code> , <code>closepass()</code> ,																

closelog()

	Close log file.
Synopsis	<pre>#include <syslog.h> void closelog ();</pre>
Description	Closes the log file opened with <code>openlog()</code> .
See Also	<code>openlog()</code> , <code>syslog()</code> , <code>vsyslog()</code>

closepass()

Prepare to pass a socket to another task

Synopsis

```
#include <socket.h>
#include <unistd.h>
unsigned long closepass ( fd )
int fd;
```

Description

Retrieves a special token about a socket. This token is used later by `openold()` to open the socket by another task.

The `closepass()` function is used by the owning task of a socket to prepare the socket to be passed to another task within the same address space. `fd` is the socket descriptor of the socket to be passed. A token is returned at successful completion of the call to `closepass`.

The token must be passed by the owning task to the receiving task via application-specific interprocess communication. Once the token has been passed to the receiving task, the owning task should use the `close()` function to close the socket. At approximately the same time, the receiving task should call `openold()` to actually receive ownership of the socket. The call to `close()` blocks until the receiving task completes its `openold()` request.

This is a sample of events that must occur in time-ordered sequence. Assume that the main task has opened a socket, using `socket()` or `accept()`:

Main Task	Subtask
1) Call <code>closepass(fd)</code> .	
2) Send IPC to subtask passing token returned by <code>closepass</code> .	
3) Call <code>close(fd)</code> . On return from <code>close</code> , main task can no longer reference this <code>fd</code> .	
	4) Receive token from main task via application-dependent IPC.
	5) Call <code>openold(token)</code> .
6) Upon return from <code>close</code> , main task may no longer reference <code>fd</code> .	
	7) Subtask may now use <code>fd</code> returned by <code>openold</code> to access network.
	8) Subtask is through with socket.
	9) Call <code>close(fd)</code> to remove socket.

Return Value	If successful, <code>connect()</code> returns a token that relates to the socket. Otherwise, the value -1 is returned, and the error code stored in the global integer <code>errno</code> indicates the nature of the error.
Error Codes	The <code>connect()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code> : EBADF The <code>fd</code> specifies an invalid descriptor.
Implementation Notes	The implementation of this function is provided to ease the development of server-oriented socket applications using the socket library.
See Also	<code>accept()</code> , <code>close()</code> , <code>openold()</code> , <code>socket()</code>

connect()

Initiate a connection on a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
int connect ( s, name, namelen )
int s;
struct sockaddr *name;
int namelen;
```

Description

Assigns the name of the peer communications endpoint. If the socket is of type `SOCK_STREAM`, a connection is established between the endpoints. If the socket is of type `SOCK_ASSOC`, a permanent association is maintained between the endpoints until changed with another `connect()` function.

Parameters:

<code>s</code>	A socket of type: <code>SOCK_ASSOC</code> - Specifies the peer with which the socket is to be associated; this address is where datagrams are to be sent and is the only address from which datagrams are to be received. <code>SOCK_STREAM</code> - Attempts to make a connection to another socket.
<code>name</code>	Specifies the other socket, which is an address in the communications domain of the socket. Each communications domain interprets the name parameter in its own way.
<code>namelen</code>	Specifies the size of the name array.

Generally, stream sockets can successfully connect() only once; datagram sockets can use connect() multiple times to change their association. Datagram sockets can dissolve the association by connecting to an invalid address, such as a null address.

Return Value If the connection or association succeeds, a value of zero is returned. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes The connect() function returns these error codes (from <errno.h>) in the global integer errno:

EADDRINUSE	The address is already in use.
EADDRNOTAVAIL	The specified address is not available. This is caused by the name specifying a remote port of zero or a remote address of INADDR_ANY (0).
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EALREADY	The socket is non-blocking and a previous connection attempt has not yet been completed.
EBADF	The argument s is not a valid descriptor.
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.
ECONNREFUSED	The attempt to connect was forcefully rejected.
ECONNRESET	The remote endpoint reset the connection request.
EDESTUNREACH	Remote destination is now unreachable.
EFAULT	The pointer name points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINPROGRESS	The socket is non-blocking and the connection cannot be completed immediately. It is possible to select() for completion by selecting the socket for writing.
EINVAL	The length of the endpoint name, namelen, does not equal the size of a sockaddr_in structure.
EISCONN	The socket is already connected and does not allow multiple connects with the same socket.
ENETDOWN	Local network interface is down.
ENETUNREACH	The network is not reachable from this host.

	EOPNOTSUPP	The socket is listening for incoming connection requests and therefore cannot also be used for outgoing connection requests.
	EOPNOTSUPP	The socket does not support the connecting or associating of two endpoints. It operates only in connectionless mode.
	ETIMEDOUT	Connection establishment timed out without establishing a connection.
Implementation Notes		Unlike UNIX sockets, this implementation does allow SOCK_ASSOC (UDP sockets using association oriented operation) to listen for and accept connections.
See Also		accept(), select(), socket(), getsockname()

fcntl()

File control.

Synopsis

```
#include <socket.h>
#include <uio.h>
int fcntl ( s, cmd, arg )
int s;
int cmd;
int arg;
```

Description

Provides control over file descriptors. Asynchronous notification and non-blocking I/O can be turned on and off using this function call.

fcntl() performs a variety of control functions on a socket, indicated by s. An fcntl cmd has encoded in it whether the argument, argp, is an input parameter supplied by the caller or an output parameter returned by the fcntl() function.

Permissible values for cmd are defined in <socket.h>. The current implementation of fcntl() supports these requests:

F_GETFD	Used to get the file descriptor flags.
F_SETFD	Used to set the file descriptor flags.
F_GETFL	Used to get the socket status flags.
F_SETFL	Used to set the socket status r flags.

The flags for F_GETFL and F_SETFL are:

FNDELAY Select non-blocking I/O; if no data is available to a read system call or if a write operation would block, the call returns a -1 with the error EWOULDBLOCK.

FASYNC Enable the SIGIO, SIGURG, and SIGPIPE signals to be sent to the process when the triggering events occur.

Return Value If fcntl() is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes The fcntl() function returns these error codes (from <errno.h>) in the global integer errno:

EBADF The *s* argument is not a valid descriptor.

EINVAL *cmd* or *arg* is invalid or not supported by this implementation of fcntl().

Implementation Notes When setting the socket descriptor flags with the command of F_SETFL, the request is destructive to the variable that saves the flags. To set both the FNDELAY and FASYNC flags, a single request must be made specifying both options.

See Also accept(), connect(), recv(), recvfrom(), recvmsg(), send(), sendmsg(), sendto(), write()

gethostbyaddr()

Get host information by address.

Synopsis

```
#include <netdb.h>
struct hostent *gethostbyaddr ( addr, len, type )
char *addr;
int len, type;
```

Description

Obtains the official name of a host when its network address is known. A name server is used to resolve the address if one is available; otherwise, the name is obtained (if possible) from a database maintained on the local system.

gethostbyaddr() returns a pointer to an object with the structure hostent. This structure contains either the information obtained from the local name server or extracted from an internal host database whose function is similar to /etc/hosts on a UNIX-based system. If the local name server is not running, gethostbyaddr() looks up the information in the internal host database.

```
struct hostent
{
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
#define h_addr h_addr_list [ 0 ]
```

Parameters:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A zero-terminated array of network addresses for the host.
h_addr	The first address in h_addr_list; this is for backward compatibility.

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently supported.

Return Value

The gethostbyaddr() function returns a pointer to the hostent structure, if successful. If unsuccessful, a null pointer is returned, and the external integer h_errno is set to indicate the nature of the error.

Error Codes	The <code>gethostbyaddr()</code> function can return these error codes in the external integer <code>h_errno</code> :	
	<code>HOST_NOT_FOUND</code>	No such host is known.
	<code>NO_ADDRESS</code>	The requested name is valid, but does not have an address in the Internet domain.
	<code>NO_RECOVERY</code>	This is a nonrecoverable error.
	<code>TRY_AGAIN</code>	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.
See Also	<code>gethostbyname()</code>	

gethostbyname()

Get host information by name.

Synopsis

```
#include <netdb.h>
struct hostent *gethostbyname ( name )
char *name;
```

Description

Obtains the network address (or list of addresses) of a host when its official name (or alias) is known. A name server is used to resolve the name if one is available; otherwise, the address is obtained (if possible) from a database maintained on the local system.

The `gethostbyname()` function returns a pointer to an object with the following structure. This structure contains either the information obtained from the local name server or extracted from an internal host database whose function is similar to `/etc/hosts` on a UNIX-based system. If the local name server is not running, `gethostbyname()` looks up the information in the internal host database.

```
struct hostent
{
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
#define h_addr h_addr_list [ 0 ]
```

Parameters:

<code>h_name</code>	Official name of the host
<code>h_aliases</code>	A zero-terminated array of alternate names for the host
<code>h_addrtype</code>	The type of address being returned; currently always <code>AF_INET</code>
<code>h_length</code>	The length, in bytes, of the address
<code>h_addr_list</code>	A zero-terminated array of network addresses for the host
<code>h_addr</code>	The first address in <code>h_addr_list</code> (this is for backward compatibility)

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

Return Value The `gethostbyname()` function returns a pointer to the `hostent` structure, if successful. If unsuccessful, a null pointer is returned, and the external integer `h_errno` is set to indicate the nature of the error.

Error Codes The `gethostbyname()` function can return these error codes in the external integer `h_errno`:

<code>HOST_NOT_FOUND</code>	No such host is known.
<code>NO_ADDRESS</code>	The requested name is valid, but does not have an address in the Internet domain.
<code>NO_RECOVERY</code>	This is a nonrecoverable error.
<code>TRY_AGAIN</code>	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.

See Also `gethostbyaddr()`

gethostname()

	Get name of local host.
Synopsis	<pre>#include <socket.h> #include <uio.h> int gethostname (name, namelen) char *name; int namelen;</pre>
Description	<p>Obtains the name of the local host. This is the official name by which other hosts in the communications domain reference it. Generally, a host belonging to multiple communication domains, or connected to multiple networks within a single domain, has one official name by which it is known.</p> <p>The <code>gethostname()</code> function returns the standard host name for the local system. The parameter <code>namelen</code> specifies the size of the name array. The returned name is null-terminated unless insufficient space is provided. Host names are limited to <code>MAXHOSTNAMELEN</code> (from <code><socket.h></code>) characters, which is currently 64.</p>
Return Value	If <code>gethostname()</code> is successful, a value of zero is returned. A return value of -1 indicates an error. See <i>Implementation Notes</i> in this section for more information.
Error Codes	<p>The <code>gethostname()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code>. If the call is successful, zero is returned to the user and the name of the local host is placed in the user-provided buffer. If unsuccessful, a minus one (-1) is returned, and the external integer <code>h_errno</code> is set to indicate the nature of the error.</p> <p>The <code>gethostname()</code> function can return, in the external integer <code>h_errno</code>, the error code <code>HOST_NOT_FOUND</code>, which indicates that the host name is not found or the <code>namelen</code> parameter is less than or equal to zero.</p>
Implementation Notes	<p>On a UNIX system this request is a system call. It obeys all of the standards for system calls to include setting <code>errno</code> when an error is detected. This implementation, however, treats this function as a network database call and uses the external variable <code>h_errno</code> to describe errors to the user.</p> <p>Note: This function does not set <code>errno</code>.</p>

getnetbyaddr()

Get network information by address.

Synopsis

```
#include <netdb.h>
struct netent *getnetbyaddr ( net, type )
long net;
int type;
```

Description

Obtains the official name of a network when the network address (or network number) is known. This information is acquired from a network database maintained by the local system.

The `getnetbyaddr()` function returns a pointer to an object with this structure containing information extracted from an internal network database. This database is similar in function to the `/etc/networks` file on UNIX-based systems:

```
struct netent
{
    char          *n_name;
    char          **n_aliases;
    int           n_addrtype;
    u_long        n_net;
};
```

Parameters:

<code>n_name</code>	The official name of the network.
<code>n_aliases</code>	A zero-terminated list of alternate names for the network.
<code>n_addrtype</code>	The type of the network number returned; currently only <code>AF_INET</code> .
<code>n_net</code>	The network number in machine byte order.

All information is contained in a static area and must be copied if it is to be saved. The `getnetbyaddr()` function sequentially searches from the beginning of the network database until a matching network address and type is found, or until the end of the database is reached. Only the Internet address format is currently supported.

Return Value

The `getnetbyaddr()` function returns a pointer to the `netent` structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes	<p>The <code>getnetbyaddr()</code> function may return these error codes in the external integer <code>h_errno</code>:</p> <p><code>NET_NOT_FOUND</code> No such network is known.</p> <p><code>NO_ADDRESS</code> The requested name is valid, but does not have an address in the Internet domain.</p> <p><code>NO_RECOVERY</code> This is a nonrecoverable error.</p> <p><code>TRY_AGAIN</code> This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.</p>
Implementation Notes	<p>The external variable <code>h_errno</code> is not set by UNIX implementations of this function.</p>
See Also	<p><code>getnetbyname()</code></p>

getnetbyname()

Get network information by name.

Synopsis

```
#include <netdb.h>
struct netent *getnetbyname ( name )
char *name;
```

Description

Obtains the network number (or list of network numbers) of a network when the official name of the network is known. This information is acquired from a network database maintained by the local system.

getnetbyname() returns a pointer to an object with this structure containing information extracted from an internal network database. This database is similar in function to the /etc/networks file on UNIX-based systems:

```
struct netent
```

```
{
    char      *n_name;
    char      **n_aliases;
    int       n_addrtype;
    u_long    n_net;
};
```

Parameters:

n_name	The official name of the network.
n_aliases	A zero-terminated list of alternate names for the network.
n_addrtype	The type of network number returned; currently always AF_INET.
n_net	The network number in machine byte order.

All information is contained in a static area and must be copied if it is to be saved. The getnetbyname() function sequentially searches from the beginning of the network database until a matching network name is found, or until the end of the database is reached. Only the Internet address format is currently understood.

Return Value

The getnetbyname() function returns a pointer to the netent structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes	The <code>getnetbyname()</code> function may return these error codes in the external integer <code>h_errno</code> : <code>NET_NOT_FOUND</code> No such network is known. <code>NO_ADDRESS</code> The requested name is valid, but does not have an address in the Internet domain. <code>NO_RECOVERY</code> This is a nonrecoverable error. <code>TRY_AGAIN</code> This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
Implementation Notes	The external variable <code>h_errno</code> is not set by UNIX implementations of this function.
See Also	<code>getnetbyaddr()</code>

getopts

	Parse command options.
Synopsis	<code>getopts optstring name [argument...]</code> <code>/usr/lib/getoptcvd [-b] filename</code> <code>/usr/lib/getoptcvd</code>
Description	Parses positional parameters and to check for valid options. <code>optstring</code> must contain the option letters the command using <code>getopts</code> will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

getpeername()

	Get name of connected peer.								
Synopsis	<pre>#include <socket.h> #include <uio.h> int getpeername (s, name, namelen) int s; struct sockaddr *name; int *namelen;</pre>								
Description	<p>Gets the name of a connected peer. The name was assigned when a connect() or accept() function was successfully completed.</p> <p>getpeername() returns the name of the peer connected to socket s. The namelen parameter should be initialized to indicate the amount of space pointed to by name. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.</p>								
Return Value	If getpeername() is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.								
Error Codes	<p>The getpeername() function returns these error codes (from <errno.h>) in the global integer errno:</p> <table><tr><td>EBADF</td><td>The argument s is not a valid descriptor.</td></tr><tr><td>EFAULT</td><td>Either the name pointer or name length pointer points to inaccessible memory.</td></tr><tr><td>EINVAL</td><td>The name length passed by the user is less than or equal to zero.</td></tr><tr><td>ENOTCONN</td><td>The socket is not connected.</td></tr></table>	EBADF	The argument s is not a valid descriptor.	EFAULT	Either the name pointer or name length pointer points to inaccessible memory.	EINVAL	The name length passed by the user is less than or equal to zero.	ENOTCONN	The socket is not connected.
EBADF	The argument s is not a valid descriptor.								
EFAULT	Either the name pointer or name length pointer points to inaccessible memory.								
EINVAL	The name length passed by the user is less than or equal to zero.								
ENOTCONN	The socket is not connected.								
See Also	accept(), bind(), socket(), getsockname()								

getprotobyname()

Get protocol information by name.

Synopsis

```
#include <netdb.h>
struct protoent *getprotobyname ( name )
char *name;
```

Description

Gets the protocol number when the official protocol name is known. This information is acquired from a protocol database maintained by the local system.

getprotobyname() returns a pointer to an object with this structure containing information extracted from an internal network protocol database. This database is similar in function to the /etc/protocols file on UNIX-based systems:

```
struct protoent
{
    char *p_name;
    char **p_aliases;
    int p_proto;
};
```

p_name	The official name of the protocol
p_aliases	A zero-terminated list of alternate names for the protocol
p_proto	The protocol number

All information is contained in a static area and must be copied if it is to be saved. The getprotobyname() function sequentially searches from the beginning of the network protocol database until a matching protocol name is found, or until the end of the database is reached.

Return Value

The getprotobyname() function returns a pointer to the protoent structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes

The getprotobyname() function may return these error codes in the external integer h_errno:

NO_ADDRESS	The requested name is valid, but does not have an address in the Internet domain.
NO_RECOVERY	This is a nonrecoverable error.
PROTO_NOT_FOUND	No such protocol is known.
TRY_AGAIN	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.

Implementation Notes	The external variable <code>h_errno</code> is not set by UNIX implementations of this function.
See Also	<code>getprotobynumber()</code>

getprotobynumber()

Get protocol information by number.

Synopsis

```
#include <netdb.h>
struct protoent *getprotobynumber ( proto )
int proto;
```

Description

Obtains the official name of a protocol when the protocol number is known. This information is acquired from a network database maintained by the local system.

`getprotobynumber()` returns a pointer to an object with this structure containing information extracted from an internal network protocol database. This database is similar in function to the `/etc/protocols` file on UNIX-based systems:

```
struct protoent
{
    char    *p_name;
    char    **p_aliases;
    int     p_proto;
};
```

Parameters:

<code>p_name</code>	The official name of the protocol
<code>p_aliases</code>	A zero-terminated list of alternate names for the protocol
<code>p_proto</code>	The protocol number

All information is contained in a static area and must be copied if it is to be saved. The `getprotobynumber()` function sequentially searches from the beginning of the network protocol database until a matching protocol number is found, or until the end of the database is reached.

Return Value

If successful, the `getprotobynumber()` function returns a pointer to the `protoent` structure. If unsuccessful, a null pointer is returned.

Error Codes	The <code>getprotobyname()</code> function may return these error codes in the external integer <code>h_errno</code> :								
	<table> <tr> <td><code>NO_ADDRESS</code></td> <td>The requested name is valid, but does not have an address in the Internet domain.</td> </tr> <tr> <td><code>NO_RECOVERY</code></td> <td>This is a nonrecoverable error.</td> </tr> <tr> <td><code>PROTO_NOT_FOUND</code></td> <td>No such protocol is known.</td> </tr> <tr> <td><code>TRY_AGAIN</code></td> <td>This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.</td> </tr> </table>	<code>NO_ADDRESS</code>	The requested name is valid, but does not have an address in the Internet domain.	<code>NO_RECOVERY</code>	This is a nonrecoverable error.	<code>PROTO_NOT_FOUND</code>	No such protocol is known.	<code>TRY_AGAIN</code>	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.
<code>NO_ADDRESS</code>	The requested name is valid, but does not have an address in the Internet domain.								
<code>NO_RECOVERY</code>	This is a nonrecoverable error.								
<code>PROTO_NOT_FOUND</code>	No such protocol is known.								
<code>TRY_AGAIN</code>	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.								
Implementation Notes	The external variable <code>h_errno</code> is not set by UNIX implementations of this function.								
See Also	<code>getprotobyname()</code>								

getservbyname()

Get service information by name.

Synopsis

```
#include <netdb.h>
struct servent *getservbyname ( name, proto )
char *name, *proto;
```

Description

Obtain the port number associated with a service when its official name is known. This information is acquired from a service database maintained by the local system.

`getservbyname()` returns a pointer to an object with this structure containing information extracted from an internal network service database. This database is similar in function to the `/etc/services` file on UNIX-based systems:

```
struct servent
{
    char    *s_name;
    char    **s_aliases;
    int     s_port;
    char    *s_proto;
};
```

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides.
s_proto	The name of the protocol to use when contacting the service.

All information is contained in a static area and must be copied if it is to be saved. The `getservbyname()` function sequentially searches from the beginning of the network service database until a matching service name is found, or until the end of the database is reached. If a protocol name is also supplied (not null), searches must also match the protocol.

Return Value The `getservbyname()` function returns a pointer to the `servent` structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes The `getservbyname()` function may return these error codes in the external integer `h_errno`:

NO_ADDRESS	The requested name is valid, but does not have an address in the Internet domain.
NO_RECOVERY	This is a nonrecoverable error.
SERV_NOT_FOUND	No such service is known.
TRY_AGAIN	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.

Implementation Notes The external variable `h_errno` is not set by UNIX implementations of this function.

See Also `getservbyport()`, `getprotobyname()`

getservbyport()

Get service information by port.

Synopsis

```
#include <netdb.h>
struct servent *getservbyport ( port, proto )
int port;
char *proto;
```

Description

Obtains the official name of a service when the port number associated with the service is known. This information is acquired from a service database maintained by the local system.

getservbyport() returns a pointer to an object with this structure containing information extracted from an internal network service database. This database is similar in function to the /etc/services file on UNIX-based systems:

```
struct servent
{
    char    *s_name;
    char    **s_aliases;
    int     s_port;
    char    *s_proto;
};
```

Parameters:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides.
s_proto	The name of the protocol to use when contacting the service.

All information is contained in a static area and must be copied if it is to be saved. The getservbyport() function sequentially searches from the beginning of the network service database until a matching port number is found, or until the end of the database is reached. If a protocol name is also supplied (not null), searches must also match the protocol.

Return Value

The getservbyport() function returns a pointer to the servent structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes	The <code>getservbyport()</code> function can return these error codes in the external integer <code>h_errno</code> : <code>NO_ADDRESS</code> The requested name is valid, but does not have an address in the Internet domain. <code>NO_RECOVERY</code> This is a nonrecoverable error. <code>SERV_NOT_FOUND</code> No such service is known. <code>TRY_AGAIN</code> This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
Implementation Notes	The external variable <code>h_errno</code> is not set by UNIX implementations of this function.
See Also	<code>getservbyname()</code> , <code>getprotobyname()</code>

getsockname()

	Get socket name.
Synopsis	<pre>#include <socket.h> #include <uio.h> int getsockname (s, name, namelen) int s; struct sockaddr *name; int *namelen;</pre>
Description	Obtains the name assigned to a socket, which is the address of the local endpoint and was assigned with a <code>bind()</code> function. <code>getsockname()</code> returns the current name for the specified socket. The <code>namelen</code> parameter should be initialized to indicate the amount of space pointed to by <code>name</code> . On return, it contains the actual size of the name returned (in bytes).
Return Value	If <code>getsockname()</code> is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer <code>errno</code> indicates the nature of the error.

Error Codes	The <code>getsockname()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code> :
	<code>EBADF</code> The argument <code>s</code> is not a valid descriptor.
	<code>EFAULT</code> Either the name pointer or name length pointer points to inaccessible memory.
	<code>EINVAL</code> The name length passed by the user is less than or equal to zero.
See Also	<code>bind()</code> , <code>socket()</code>

getsockopt()

Get options on a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
int getsockopt ( s, level, optname, optval, optlen )
int s, level, optname;
char *optval;
int *optlen;
```

Description

Retrieves options currently associated with a socket. Options always exist at the socket level and can also exist at layers within the underlying protocols. Options are set with the `setsockopt()` function. Options can exist at multiple protocol levels; they are always present at the uppermost socket level.

When retrieving socket options, the level at which the option resides and the name of the option must be specified. To retrieve options at the socket level, `level` is specified as `SOL_SOCKET`. To retrieve options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be returned by the TCP protocol, `level` should be set to the protocol number of TCP. Read [getprotobyname\(\)](#) for additional information.

The parameters `optval` and `optlen` identify a buffer in which the value for the requested option is to be returned. The `optlen` parameter is a value-result parameter, initially containing the size of the buffer pointed to by `optval` and modified on return to indicate the actual size of the value returned. If no option value is to be returned, `optval` can be supplied as zero.

The `optname` parameter is passed uninterpreted to the appropriate protocol module for interpretation. The include file `<socket.h>` contains definitions for socket level options, described in the following table. Options at other protocol levels vary in format and name; consult the appropriate appendix for additional information.

Most socket-level options return an int parameter for optval. For boolean options, a non-zero value indicates the option is enabled, and a zero value indicates the option is disabled. SO_LINGER uses a struct linger parameter, defined in <socket.h>, which specifies the desired state of the option and the linger interval.

The following table lists options that are recognized at the socket level. Except as noted, each can be examined with getsockopt() and set with setsockopt():

Socket Option	Description
SO_BROADCAST	Requests permission to send broadcast datagrams on the socket.
SO_DEBUG	Enables debugging in the socket modules. This option is implemented slightly differently than UNIX, in that, on UNIX it allows for debugging at the underlying protocol modules.
SO_DONTROUTE	Indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_ERROR	Returns any pending error on the socket and clears the error status. This option can be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.
SO_KEEPALIVE	<p>The keepalive option enables the periodic probing of the remote connection. This option specifies the type of keepalive to use. It also turns off the keepalive option. Should the connected peer fail to respond to the probe, the connection is considered broken and the process using the socket is notified via errno ETIMEDOUT. These int values are supported:</p> <ul style="list-style-type: none"> 0 - Turn off keepalive 1 - Use keepalive with no data, and do not abort the session if no response 2 - Use keepalive with no data, and abort the session if no response 3 - Use keepalive with data. <p>See the notes following this section for a discussion of keepalive. Their use is discouraged and is provided only for those applications that are incapable of detecting idle sessions.</p>
SO_LINGER	Controls the action taken when unsent messages are queued on socket and a close() is performed. If the socket promises reliable delivery of data and SO_LINGER is set to a value other than zero, the system blocks the process on the close() attempt until it is able to transmit the data or until the number of seconds specified by the SO_LINGER option expire or until the connection has timed out. If SO_LINGER is set and the interval is zero, the socket is closed once the system has scheduled that an orderly release be performed.
SO_MAXRCVBYTCNT	Returns the maximum allowable setting for the number of bytes of buffering allowed for the receive simplex of the socket. The SO_RCVBYTCNT option cannot be set to a value larger than that returned by this option.

Socket Option	Description
SO_MAXRCVREQCNT	Returns the maximum allowable setting for the number of outstanding receive requests. The SO_RCVREQCNT option cannot be set to a value larger than that returned by this option.
SO_MAXSNDBYTCNT	Returns the maximum allowable setting for the number of bytes of buffering allowed for the transmit simplex of the socket. The SO_SNDBYTCNT option cannot be set to a value larger than that returned by this option.
SO_MAXSNDRREQCNT	Returns the maximum allowable setting for the number of outstanding transmit requests. The SO_SNDREQCNT option cannot be set to a value larger than that returned by this option.
SOO_OBINLINE	This option is not currently supported by this release of the socket library. It is implemented on UNIX, and its purpose is to request that out-of-band data be placed in the normal data input queue as received by protocols that support out-of-band data; it is then be accessible with <code>recv()</code> or <code>read()</code> functions without the <code>MSG_OOB</code> flag.
SO_OPTIONS	Returns the current setting of all socket level options as contained in the socket's control block. This option is specific to this implementation and is not portable to other socket libraries. This option was designed for debugging and is not an option to be used by the average user. The socket options are defined in the header file <code>sockvar.h</code> .
SO_RCVBUF	Adjusts the normal buffer size allocated for input. The buffer size can be increased for high-volume connections to improve throughput or can be decreased to limit the possible backlog of incoming data. The system places an absolute limit on this value. This implementation of sockets provides for this option for backward compatibility, but also allows for buffer options that are more specific to the underlying API and therefore provide a better method of controlling a socket's buffering characteristics. These options are <code>SO_RCVBYTCNT</code> and <code>SO_RCVREQCNT</code> . All buffering options can be set only once by the socket user and must be done before any data is sent or received on the socket.
SO_RCVBYTCNT	Adjusts the number of bytes allocated to the receive circular buffer for a socket. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the receive byte count of the circular buffer is changed, it must be done prior to sending or receiving any data on the socket.
SO_RCVREQCNT	Adjusts the number of receive requests that can be active within the socket library at a given time. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the receive request count is changed, it must be done prior to sending or receiving any data on the socket.

Socket Option	Description
SO_READFRAG	This option lets a user of a datagram type socket that preserves message boundaries read a datagram a piece at a time. Traditionally with UNIX sockets, if a user issues a read request for 100 bytes and the datagram being read consists of 120 bytes, the socket returns the first 100 bytes to the caller and then flushes the remaining 20 bytes. This default method of operation by this implementation of sockets can be overridden with this option. This option does not allow parsing of pieces of a single datagram into a single user buffer. When this option is used, the user must determine the boundaries of datagrams. This option is specific to this implementation and is not portable to other socket implementations.
SO_REUSEADDR	This option indicates that the rules used in validating addresses supplied in a bind() function should allow reuse of local addresses.
SO_SENDALL	This option guarantees that any form of send request (send(), sendto(), sendmsg(), write(), or writev()) that is done in a blocking mode transmits all the data specified by the user. Traditionally BSD sockets would send as many bytes as the current buffering allocation allowed and then return to the user with a count of the actual number of bytes transmitted. If the user requested that a write of 200 bytes be done, but there currently was only buffering space for 150 bytes, the socket would queue 150 bytes for transmission and return a count of 150 to the caller to indicate that 50 bytes could not be transmitted due to lack of buffer space. This implementation of sockets acts identically to a UNIX socket under the same scenario under the default setting of the socket options. However, if this option is turned ON in Unicenter TCPaccess sockets, the socket blocks the user until all of the data has been queued for transmission or some type of error occurs. This option is specific to this implementation
SO_SNDBUF	Adjusts the normal buffer size allocated for output. The buffer size can be increased for high-volume connections to improve throughput or can be decreased to limit the possible backlog of outgoing data. The system places an absolute limit on this value. This implementation of sockets provides this option for backward compatibility. It also allows for buffer options that are more specific to the underlying API and therefore provides a better method of controlling a socket's buffering characteristics. These options are SO_SNDBYTCNT and SO_SNDREQCNT. All buffering options can be set only once by the socket user and must be done before any data is sent or received on the socket.
SO_SNDBYTCNT	Adjusts the number of bytes allocated to the send circular buffer for a socket. This option is specific to this implementation of sockets and is not portable to other socket libraries. This option can be set only once successfully, and if the send byte count of the circular buffer is changed, it must be done prior to sending or receiving any data on the socket.

Socket Option	Description
SO_SNDREQCNT	Adjusts the number of send requests that can be active within the socket library at a given time. This option is specific to this implementation of sockets and is not portable to other socket libraries. This option can be set only once successfully, and if the send request count is changed, it must be done prior to sending or receiving any data on the socket.
SO_STATE	Returns the current socket state as contained in the socket's control block. This option is specific to this implementation and is not portable to other socket libraries. This option was designed for debugging and is not an option the average user should use. The socket states are defined in the header file <code>sockvar.h</code> .
SO_SUBSTATE	Returns the current setting of the socket's substate as contained in the socket's control block. This option is specific to this implementation and is not portable to other socket libraries. This option is for debugging and is not for use by the average user. The socket substates are defined in the header file <code>sockvar.h</code> .
SO_TYPE	Returns the type of the socket, such as <code>SOCK_STREAM</code> ; it is useful for servers that inherit sockets on startup.
SO_USELOOPBACK	Requests that the loopback interface is used rather than a real physical interface. These options are recognized at the TCP level (<code>IPPROTO_TCP</code>):
TCP_MAXSEG	This option is not supported by this implementation of sockets at the present release. On UNIX this option lets the user of a <code>SOCK_STREAM</code> socket declare the value of the maximum segment size for TCP to use when negotiating this value with its remote endpoint.
TCP_NODELAY	Ensures that TCP type sockets (<code>SOCK_STREAM</code>) send data as soon as possible and do not wait for more data or a given amount of time to enhance the packetizing algorithm. This option is similar to the BSD UNIX socket option.
Return Value	If <code>getsockopt()</code> is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer <code>errno</code> indicates the nature of the error.

Socket Option	Description
Error Codes	The <code>getsockopt()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code> :
EBADF	The <code>s</code> argument is not a valid descriptor.
EFAULT	The pointer to the value buffer points to inaccessible memory.
EFAULT	The pointer to the value buffer length points to inaccessible memory.
EINVAL	The size of the value buffer does not equal the size of the option. Most options require an integer length buffer or, in the case of <code>SO_LINGER</code> , the buffer must be the size of the <code>linger</code> structure.
EINVAL	The option buffer size is greater than the maximum allowed by the API.
EINVAL	The option is not supported at the level requested.
EINVAL	No options can be read from the protocol layers.
Implementation Notes	These options are recognized at the socket level on BSD UNIX systems, but are not supported by the API. If any of these options are referenced, an error is generated:
SO_SNDLOWAT	Sets the send low water buffering mark.
SO_RCVLOWAT	Sets the receive low water buffering mark.
SO_SNDTIMEO	Sets the send timeout value. This option is not currently implemented in UNIX.
SO_RCVTIMEO	Sets the receive timeout value. This option is not currently implemented in UNIX.
See Also	<code>ioctl()</code> , <code>setsockopt()</code> , <code>socket()</code> , <code>fcntl()</code>

getstablesize()

	Get socket table size.
Synopsis	<pre>#include <socket.h> #include <uio.h> int getstablesize()</pre>
Description	<p>Provides a method by which the socket library user can determine the maximum number of sockets that can be used at any given time.</p> <p>The <code>getstablesize()</code> function performs the same function as the <code>gettablesize</code> function call on UNIX. It varies somewhat in that it only returns the maximum number of sockets that can be used at a given time and not the combined total of socket and file descriptors.</p>
Implementation Notes	See the function description for the differences.

gettimeofday()

	Get the date and time.
Synopsis	<pre>#include <sys/time.h> int gettimeofday (tp) struct time_t *tp;</pre>
Description	<p>Gets the system's notion of the current time. The current time is expressed in elapsed seconds and microseconds since a particular time. The time chosen depends on <code>#define USESASCTIME</code>. If this is defined, <code>time_t</code> will be returned as a double and is relative to 00:00 January 1, 1900. If the above define is not used, <code>time_t</code> is returned as an int relative to 00:00 January 1, 1970.</p>
Implementation Notes	Consult the SAS/C documentation for more information on the <code>time_t</code> structure.

htonl()

Convert long values from host to network byte order.

Synopsis

```
#include <inet.h>
unsigned long htonl ( hostlong );
unsigned long hostlong;
```

Description

Maintains compatibility with application programs ported from other systems that byte-swap memory. These systems store 32-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols.

htonl() converts 32-bit quantities from host byte order to network byte order. This function was originally provided for UNIX systems running on VAX processors that byte-swap memory. On machines such as IBM mainframes where network and host byte order are the same, this function is defined as a null macro in the include file <inet.h>.

This function is most often used in conjunction with Internet addresses and port numbers as returned by gethostbyname() and getservbyname().

Implementation Notes

The htonl() function is implemented as a null macro and performs no operation on the function argument.

See Also

gethostbyname(), getservbyname()

htons()

Convert short values from host to network byte order.

Synopsis

```
#include <inet.h>
unsigned short htons ( hostshort );
unsigned short hostshort;
```

Description

Maintains compatibility with application programs ported from other systems that byte-swap memory. These systems store 16-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols.

htons() converts 16-bit quantities from host byte order to network byte order. This function was originally provided for UNIX systems running on VAX processors that byte-swap memory. On machines such as IBM mainframes, where network and host byte order are the same, this function is defined as a null macro in the include file <inet.h>.

This function is most often used in conjunction with Internet addresses and port numbers as returned by gethostbyname() and getservbyname().

Implementation Notes

The htons() function is implemented as a null macro and performs no operation on the function argument.

See Also

gethostbyname(), getservbyname()

inet()

Internet address manipulation routines.

Synopsis

```
#include <socket.h>
#include <uio.h>
#include <inet.h>
unsigned long inet_addr ( cp )
char *cp;
unsigned long inet_network ( cp )
char *cp;
char *inet_ntoa ( in )
struct in_addr in;
unsigned long inet_makeaddr ( net, lna )
int net, lna;
int inet_lnaof ( in )
struct in_addr in;
int inet_netof ( in )
struct in_addr in;
```

Description

A set of routines that construct Internet addresses or break Internet addresses down into their component parts. Routines that convert between the binary and ASCII ("dot" notation) form of Internet addresses are included.

The functions `inet_addr()` and `inet_network()` each interpret character strings representing numbers expressed in the Internet standard dot (".") notation, returning numbers suitable for use as Internet addresses and network numbers, respectively. The function `inet_ntoa()` takes an Internet address and returns an ASCII string representing the address in dot notation. The function `inet_makeaddr()` takes an Internet network number and a local network address (host number) and constructs an Internet address from it. The functions `inet_netof()` and `inet_lnaof()` break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network byte order. All network numbers and local address parts are returned as integer values in host byte order. On machines such as IBM mainframes, network and host byte order are the same.

Values specified using the Internet standard dot notation take one of these forms:

a.b.c.d

a.b.c

a.b

- When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.
- When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as 128.net.host.
- When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as net.host.
- When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in a dot notation can be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading zero implies octal; if there is no leading zero, the number is interpreted as decimal).

Return Value

The `inet_addr()` function returns an Internet address if successful, or a value of -1 if the request was unsuccessful.

The `inet_network()` function returns a network number if successful, or a value of -1 if the request was malformed.

The `inet_ntoa()` function returns a pointer to an ASCII string giving an Internet address in dot notation, `inet_makeaddr()` returns an Internet address, and `inet_netof()` and `inet_lnaof()` each return an integer value.

Implementation Notes

The function `inet_makeaddr()` returns a structure of type `in_addr` with the UNIX function. In this implementation, it returns a value of type unsigned long.

See Also

`gethostbyname()`, `getnetbyname()`

inet_aton()

Convert ASCII string to network address.

Synopsis

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>
int inet_aton( cp , in)
const char *cp;
struct in_addr *in;
```

Description

Converts an ASCII representation of an Internet Address to its Network Internet Address.

inet_aton() interprets a character string representing numbers expressed in the Internet standard '.' notation, returning a number suitable for use as an Internet address.

Implementation Notes

Values specified using the '.' (dot) notation take one of the following forms:

a.b.c.d

a.b.c

a.b

a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host". When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host". When only one part is given, the value is stored directly in the network address without any byte rearrangement. All numbers supplied as "parts" in a '.' notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; a leading zero implies octal; otherwise, the number is interpreted as decimal).

See Also

inet_ntoa()

inet_ntoa()

Convert network internet address to ASCII representation.

Synopsis

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>
char *inet_ntoa ( in )
const struct in_addr in;
```

Description

Converts an Internet network address to its ASCII “d.d.d.d” representation.

inet_ntoa() returns a pointer to a string in the base 256 notation "d.d.d.d". Internet addresses are returned in network order (bytes ordered from left to right).

Implementation Notes

inet_ntoa() points to a buffer which is overwritten on each call.

See Also

inet_aton()

ioctl()

Control I/O.

Synopsis

```
#include <socket.h>
#include <uio.h>
int ioctl ( s, request, argp )
int s;
unsigned long request;
char *argp;
```

Description

Manipulates I/O controls associated with a socket. In particular, the sensing of out-of-band data marks and the enabling of non-blocking I/O can be controlled.

The `ioctl()` function performs a variety of control functions on a socket, indicated by `s`. An `ioctl` request has encoded in it whether the argument pointed to by `argp` is an input parameter supplied by the caller or an output parameter returned by the `ioctl()` function. `request` also encodes in bytes the length of the argument.

Permissible values for `request` are defined in `<socket.h>`. The following table lists the requests supported by the current implementation of `ioctl()`:

Request	Definition
FIOASYNC	Used to enable the usage of the user-added signals of the socket library. Asynchronous events happening at the local endpoint initiate the triggering of a signal. SIGIO, SIGURG and SIGPIPE are all supported by the implementation.
FIONBIO	Used to set and clear non-blocking I/O mode. When a socket is in non-blocking I/O mode, any function that would otherwise block for flow control or synchronization completes immediately with an error. A return value of -1 indicates the error, and the error code EWOULDBLOCK stored in the external integer <code>errno</code> indicates the blocking condition. The function should be reissued later.
FIONREAD	Returns the status of the received data flag. Unlike UNIX, which actually returns the number of data bytes in the receive buffer, a return value of 1 indicates that there is data to be read. A value of zero indicates an absence of data to be read.
SIOCADDRT	Add a single routing table entry Note: Not allowed for user application programs.
SIOCATMARK	Indicates whether or not the current read pointer points to the location at which out-of-band data was received. Whenever out-of-band data is received, the location in the input stream is marked. The boolean value returned by this request indicates when the read pointer has reached the mark.
SIOCDELRT	Delete a single routing table entry. Note: Not allowed for user application programs.

Request	Definition
SIOCGIFADDR	Get interface address. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFBRDADDR	Get broadcast address. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFCONF	Get interface configuration list. This request takes an <code>ifconf</code> structure as a value-result parameter. The <code>ifc_len</code> field should be initially set to the size of the buffer pointed to by <code>ifc_buf</code> . On return, it will contain the length, in bytes of the configuration list. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFDSTADDR	Get point-to-point address for the interface. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFFLAGS	Get the interface flags. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFHWADDR	Get the hardware address. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFMETRIC	Get the metric associated with the interface. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFMTU	Get the maximum transmission unit size for interface. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFNETMASK	Get the network address mask. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFNUM	Get the number of interfaces. This request returns an integer which is the number of interface descriptions (<code>struct ifreq</code> , found in <code>if.h</code>) that will be returned by the <code>SIOCGIFCONF</code> <code>ioctl</code> ; that is, it gives an indication of how large <code>ifc_len</code> has to be.
SIOCSIFMETRIC	Sets the network interface routing metric. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .

Return Value If `ioctl()` is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes The `ioctl()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

EBADF	The <code>s</code> argument is not a valid descriptor.
EFAULT	The pointer <code>argp</code> is invalid.
EINVAL	Request or <code>argp</code> is invalid or not supported by this implementation of <code>ioctl()</code> .

Request	Definition
Implementation Notes	<p>The <code>ioctl()</code> function as implemented on UNIX-based systems is used to manipulate controls that tend to be specific to a given implementation of a protocol and often do not translate directly to other environments. As such, most of the request values defined for UNIX have no meaning in the MVS environment or are not readily portable.</p> <p>Note the difference of <code>FIONREAD</code> as previously described.</p>
See Also	<code>accept()</code> , <code>connect()</code> , <code>recv()</code> , <code>recvfrom()</code> , <code>recvmsg()</code> , <code>send()</code> , <code>sendmsg()</code> , <code>sendto()</code> , <code>write()</code>

listen()

	<p>Listen for connections on a socket.</p>
Synopsis	<pre>#include <socket.h> #include <uio.h> int listen (s, backlog) int s, backlog;</pre>
Description	<p>Indicates the application program is ready to accept connection requests arriving at a socket of type <code>SOCK_STREAM</code>. The connection request is queued (if possible) until accepted with an <code>accept()</code> function.</p> <p>To accept connections, a socket is first created with <code>socket()</code>, a readiness to accept incoming connections, and a queue limit for incoming connections, which are specified with <code>listen()</code>, and then the connections are accepted with <code>accept()</code>. The <code>listen()</code> function applies only to sockets of type <code>SOCK_STREAM</code> or <code>SOCK_ASSOC</code>.</p> <p>The <code>backlog</code> parameter defines the maximum number of pending connections that may be queued. If a connection request arrives with the queue full, the client can receive an error with an indication of <code>ECONNREFUSED</code>, or, if the underlying protocol supports retransmission, the request can be ignored so that retries can succeed.</p>
Return Value	<p>If <code>listen()</code> is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer <code>errno</code> indicates the nature of the error.</p>

Error Codes	<p>The <code>listen()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code>:</p> <p><code>EADDRINUSE</code> The specified address is not available. Another socket is currently using this endpoint to listen.</p> <p><code>EALREADY</code> The socket is currently listening.</p> <p><code>EBADF</code> The argument <code>s</code> is not a valid descriptor.</p> <p><code>EINVAL</code> The backlog variable passed by the user is less than or equal to zero.</p> <p><code>EINVAL</code> The socket is not in the bound state.</p> <p><code>EOPNOTSUPP</code> The socket is not of a type that supports the operation <code>listen()</code>.</p>
Implementation Notes	<p>The backlog can be limited to a value smaller than the current maximum of five supported by most BSD UNIX implementations. If the underlying protocol cannot support the value specified, a smaller value is substituted. <code>backlog</code> only limits the number of connection requests that can be queued simultaneously and not the total number of connections that can be accepted. A <code>listen</code> count of less than or equal to zero is invalid (<code>EINVAL</code>).</p> <p>This implementation lets the user program listen on <code>SOCK_ASSOC</code> sockets and accept the incoming connection requests.</p>
See Also	<code>accept()</code> , <code>connect()</code> , <code>socket()</code>

mvsselect()

Synchronous I/O multiplexing with optional ECB list.

Synopsis

```
#include <socket.h>
#include <uio.h>
int mvsselect ( nfd, readfds, writefds, exceptfds, timeout, ecbcount, ecblistp )
int nfd;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
int ecbcount;
unsigned long **ecblistp;
FD_SET ( fd, &fdset )
FD_CLR ( fd, &fdset )
FD_ISSET ( fd, &fdset )
FD_ZERO ( &fdset )
int fd;
fd_set fdset;
```

Description

Synchronizes processing of several sockets operating in non-blocking mode and other system events related to an optional ECB list. Sockets that are ready for reading or writing, or that have a pending exceptional condition can be selected, as well as the posting of any ECBs in the ECB list. If no sockets are ready for processing or no ECBs are posted, the mvsselect() function can block indefinitely or wait for a specified period of time (which may be zero) and then return.

mvsselect() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfd descriptors are checked in each set (the descriptors from 0 through nfd-1 in the descriptor sets are examined). Also, the ECB list is checked to see which have been posted. Only ecbcount number of ECBs are checked. On return, mvsselect() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets and ECBs posted are returned.

Sockets use MVS STIMER services. Socket applications may use up to fifteen STIMER calls per task control block (TCB). STIMER CANCEL ID=ALL must not be used by socket applications.

Note: If you invoke mvsselect() before any sockets are opened, an error code of EBADF will be returned because the internal user work area has not been initialized. A workaround for this would be to open a *throwaway* socket prior to calling mvsselect().

The descriptor sets are stored as bit fields in arrays of integers. These macros are provided for manipulating such descriptor sets:

FD_ZERO(&fdset)	Initializes a descriptor set fdset to the null set.
FD_SET(fd, &fdset)	Includes a particular descriptor fd in fdset.
FD_CLR(fd, &fdset)	Removes fd from fdset.
FD_ISSET(fd, &fdset)	Is non-zero if fd is a member of fdset, zero otherwise.

The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to FD_SETSIZE, which normally is at least equal to the maximum number of descriptors supported by the system.

If timeout is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a zero pointer, mvselect blocks indefinitely. To affect a poll, the timeout argument should be non-zero, pointing to a zero-valued timeval structure. The mvselect function uses the OS timer. If this is undesirable, the timeout variable should be a zero pointer and one of the ECBs in the list should be posted by the application-specific timeout routine.

Any of readfds, writefds, and exceptfds can be given as zero pointers if no descriptors are of interest.

Note: For release 3.1 and higher, note that the select() and mvselect() functions have changed slightly for SAS/C users. SAS/C allows signals to be raised. select() may complete if the signal is raised without the conditions on the select() parameters having been met. This new feature allows you to use signal functionality, issue a blocking receive, and do a send from a signal handler.

Return Value

If successful, mvselect() returns the sum of the number of ready descriptors that are contained in the descriptor sets and the posted ECBs, or zero if the time limit expires. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error. If mvselect() returns with an error, including one due to an interrupted call, the descriptor sets is unmodified.

Error Codes	<p>The mvselect() function returns these error codes (from <errno.h>) in the global integer errno:</p> <p>EBADF One of the descriptor sets specified an invalid descriptor.</p> <p>EINVAL The specified time limit is invalid. One of its components is negative or too large.</p> <p>EINVAL The count of socket descriptors to check is less than or equal to zero.</p> <p>EINVAL No valid socket descriptors were referenced by the three sets of masks passed by the caller.</p> <p>ENOMEM Could not allocate space to build ECB list.</p>
Implementation Notes	<p>The implementation of mvselect() provided with the API supports only MVS ECBs and descriptors associated with sockets.</p>
See Also	<p>accept(), connect(), read(), write(), recv(), send(), recvfrom(), sendto(), recvmsg(), sendmsg(), select()</p>

openold()

Open a socket passed from another task.

Synopsis

```
#include <socket.h>
#include <uio.h>
int openold(token)
unsigned long token;
```

Description

Receives the ownership of a socket from another task within the same address space.

openold() creates a new socket descriptor using a token passed from another task within the same address space. Initially, the other task had created a socket using either socket() or accept(). The other task then called closepass() and passed the token returned by closepass() via application-dependent IPC to the receiving task or subtask.

The other task then calls close() and the receiving task calls openold(). When openold() completes, the receiving task now owns the socket and the other task can no longer reference it.

The sequence of events is shown next:

Main Task	Subtask
1. Call closepass(fd).	
2. Send IPC to subtask passing token returned by closepass	
3. Call close(fd). On return from close, main task can no longer reference this fd.	
	4. Receive token from main task via application-dependent IPC.
	5. Call openold(token).
	6. Subtask may now use fd returned by openold to access network.
	7. Subtask is through with socket.
	8. Call close(fd) to remove socket.

Return Value

On successful completion, openold() returns a new socket descriptor. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes	The <code>openold()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code> :														
	<table><tr><td><code>EACCESS</code></td><td>Permission to create a socket of the specified type and/or protocol is denied.</td></tr><tr><td><code>ECONFIG</code></td><td>Socket configuration has an error, or a user session with the underlying API cannot be opened.</td></tr><tr><td><code>EINVAL</code></td><td>Token is invalid.</td></tr><tr><td><code>EMFILE</code></td><td>The system file table is full.</td></tr><tr><td><code>EMFILE</code></td><td>A new socket cannot be opened due to an API resource shortage or user endpoint allocation limit.</td></tr><tr><td><code>ENOBUFS</code></td><td>Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.</td></tr><tr><td><code>EPROTONOSUPPORT</code></td><td>The specified protocol is not supported within this domain.</td></tr></table>	<code>EACCESS</code>	Permission to create a socket of the specified type and/or protocol is denied.	<code>ECONFIG</code>	Socket configuration has an error, or a user session with the underlying API cannot be opened.	<code>EINVAL</code>	Token is invalid.	<code>EMFILE</code>	The system file table is full.	<code>EMFILE</code>	A new socket cannot be opened due to an API resource shortage or user endpoint allocation limit.	<code>ENOBUFS</code>	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.	<code>EPROTONOSUPPORT</code>	The specified protocol is not supported within this domain.
<code>EACCESS</code>	Permission to create a socket of the specified type and/or protocol is denied.														
<code>ECONFIG</code>	Socket configuration has an error, or a user session with the underlying API cannot be opened.														
<code>EINVAL</code>	Token is invalid.														
<code>EMFILE</code>	The system file table is full.														
<code>EMFILE</code>	A new socket cannot be opened due to an API resource shortage or user endpoint allocation limit.														
<code>ENOBUFS</code>	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.														
<code>EPROTONOSUPPORT</code>	The specified protocol is not supported within this domain.														
Implementation Notes	The implementation of this function is provided to ease the development of server-oriented socket applications using the socket library.														
See Also	<code>accept()</code> , <code>close()</code> , <code>closepass()</code> , <code>socket()</code>														

ntohl()

Convert long values from network to host byte order.

Synopsis

```
#include <inet.h>
unsigned long ntohl ( netlong );
unsigned long netlong;
```

Description

Maintains compatibility with application programs ported from other systems that byte-swap memory. These systems store 32-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols.

ntohl() converts 32-bit quantities from network byte order to host byte order. This function was originally provided for UNIX systems running on VAX processors that byte-swap memory. On machines such as IBM mainframes where network and host byte order are the same, this function is defined as a null macro in the include file <inet.h>.

This function is most often used in conjunction with Internet addresses and port numbers as returned by gethostbyname() and getservbyname().

Implementation Notes

The ntohl() function is implemented as a null macro and performs no operation on the function argument.

See Also

gethostbyname(), getservbyname()

ntohs()

Convert short values from network to host byte order.

Synopsis

```
#include <inet.h>
unsigned short ntohs ( netshort );
unsigned short netshort;
```

Description

Maintains compatibility with application programs ported from other systems that byte-swap memory. These systems store 16-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols.

ntohs() converts 16-bit quantities from network byte order to host byte order. This function was originally provided for UNIX systems running on VAX processors that byte-swap memory. On machines such as IBM mainframes where network and host byte order are the same, this function is defined as a null macro in the include file <inet.h>.

This function is most often used in conjunction with Internet addresses and port numbers as returned by gethostbyname() and getservbyname().

Implementation Notes

The ntohs() function is implemented as a null macro and performs no operation on the function argument.

See Also

gethostbyname(), getservbyname()

openlog()

Synopsis

```

Initialize the log file
#include <syslog.h>
#include <socket.h>
#include <inet.h>
void openlog ( ident, logopt, facility )
char *ident;
int logopt;
int facility;

```

Description

Initializes the log file.

If special processing is needed, the `openlog()` function initializes the log file. The parameter `ident` is a string that is prepended to every message. `logopt` is a bit field indicating logging options.

The following table lists the values for `logopt`:

logopt Values	Definition
LOG_PID	Log the process ID with each message. This is useful for identify in specific processes.
LOG_CONS	Write messages to the system console if they cannot be sent to syslog.
LOG_NDELAY	Open the connection to <code>syslog()</code> immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.
LOG_NOWAIT	Do not wait for processes that have been forked to log messages onto the console. This option should be used by processes that enable notification of child termination using <code>SIGCHILD</code> since <code>syslog()</code> may otherwise block waiting for a process whose exit status has already been collected. The facility parameter encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded:
LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as <code>ftpd(1M)</code> .
LOG_AUTH	The authorization system: <code>login(1)</code> , <code>su(1M)</code> , <code>getty(1M)</code> , etc.
LOG_LPR	The line printer spooling system: <code>lpr(1B)</code> , <code>lpc(1B)</code> , etc.
LOG_NEWS	Reserved for the USENET network news system.

logopt Values	Definition
LOG_UUCP	Reserved for the UUCP system; it does not currently use syslog.
LOG_CRON	The cron/at facility; crontab(1), at(1), cron(1M), etc.
LOG_LOCAL0	Reserved for local use.
LOG_LOCAL1	Reserved for local use.
LOG_LOCAL2	Reserved for local use.
LOG_LOCAL3	Reserved for local use.
LOG_LOCAL4	Reserved for local use.
LOG_LOCAL5	Reserved for local use.
LOG_LOCAL6	Reserved for local use.
LOG_LOCAL7	Reserved for local use.

See Also syslog(), closelog(), vsyslog()

read()

	Read input.																				
Synopsis	<pre>#include <socket.h> #include <uio.h> int read (s, buf, nbytes) int s; char *buf; int nbytes;</pre>																				
Description	<p>Inputs data from a socket. It operates in the same manner as reading data from a file. In a normal UNIX environment, where socket and file I/O are integrated, the read() function can be called with either a socket or file descriptor.</p> <p>read() attempts to read nbytes of data from the socket referenced by the descriptor s into the buffer pointed to by buf. On successful completion, read() returns the number of bytes actually read and placed in the buffer.</p>																				
Return Value	<p>If successful, read() returns the number of bytes read. A value of zero is returned if an end-of-file condition exists, indicating no more read() functions should be issued to this socket. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.</p>																				
Error Codes	<p>The read() function returns these error codes (from <errno.h>) in the global integer errno:</p> <table> <tr> <td>EBADF</td> <td>An invalid descriptor was specified.</td> </tr> <tr> <td>ECONNABORTED</td> <td>The incoming connection request was aborted by the remote endpoint.</td> </tr> <tr> <td>ECONNREFUSED</td> <td>The remote endpoint refused to continue the connection.</td> </tr> <tr> <td>ECONNRESET</td> <td>The remote endpoint reset the connection request.</td> </tr> <tr> <td>EDESTUNREACH</td> <td>Remote destination is now unreachable.</td> </tr> <tr> <td>EFAULT</td> <td>The buffer passed by the user points to inaccessible memory.</td> </tr> <tr> <td>EFAULT</td> <td>The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.</td> </tr> <tr> <td>EFAULT</td> <td>The pointer to the length of the name buffer points to inaccessible storage.</td> </tr> <tr> <td>EHOSTUNREACH</td> <td>Remote host is now unreachable.</td> </tr> <tr> <td>EINVAL</td> <td>The number of bytes in the receive buffer is less than zero.</td> </tr> </table>	EBADF	An invalid descriptor was specified.	ECONNABORTED	The incoming connection request was aborted by the remote endpoint.	ECONNREFUSED	The remote endpoint refused to continue the connection.	ECONNRESET	The remote endpoint reset the connection request.	EDESTUNREACH	Remote destination is now unreachable.	EFAULT	The buffer passed by the user points to inaccessible memory.	EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.	EFAULT	The pointer to the length of the name buffer points to inaccessible storage.	EHOSTUNREACH	Remote host is now unreachable.	EINVAL	The number of bytes in the receive buffer is less than zero.
EBADF	An invalid descriptor was specified.																				
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.																				
ECONNREFUSED	The remote endpoint refused to continue the connection.																				
ECONNRESET	The remote endpoint reset the connection request.																				
EDESTUNREACH	Remote destination is now unreachable.																				
EFAULT	The buffer passed by the user points to inaccessible memory.																				
EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.																				
EFAULT	The pointer to the length of the name buffer points to inaccessible storage.																				
EHOSTUNREACH	Remote host is now unreachable.																				
EINVAL	The number of bytes in the receive buffer is less than zero.																				

EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero.
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

See Also `recv()`, `recvfrom()`, `recvmsg()`, `select()`, `socket()`, `write()`

readv()

	<p>Read vectored input.</p>
Synopsis	<pre>#include <socket.h> #include <uio.h> int readv (s, iov, iovcn t) int s; struct iovec *iov; int iovcnt;</pre>
Description	<p>Inputs data from a socket in scatter mode when the input is to be noncontiguous. It operates in the same manner as reading data from a file. In a normal UNIX environment, where socket and file I/O are integrated, the readv() function can be called with either a socket or file descriptor.</p> <p>The readv() function performs the same action as read(), but scatters the input data into the iovcnt buffers specified by the members of the iov array:</p> <pre>iov[0], iov[1], ..., iov[iovcnt-1]</pre> <p>The iovec structure is defined in this way:</p> <pre>struct iovec { caddr_t iov_base; int iov_len; };</pre> <p>Each iovec entry specifies the base address and length of an area in memory where data should be placed. The readv() function always fills an area completely before proceeding to the next iov entry. On successful completion, readv() returns the total number of bytes read.</p>
Return Value	<p>If successful, readv() returns the total number of bytes read. A value of zero is returned if an end-of-file condition exits, indicating no more readv() functions should be issued to this socket. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.</p>

Error Codes	The readv() function returns these error codes (from <errno.h>) in the global integer errno:
EBADF	An invalid descriptor was specified.
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTUNREACH	The remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.
EFAULT	The pointer to the length of the name buffer points to inaccessible storage.
EFAULT	The pointer to the iovec structure points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes in the receive buffer is less than zero.
EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero.
EINVAL	The count of iovec elements in the array is greater than 16 or less than or equal to zero.
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.

	ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
	ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
	ETIMEDOUT	The connection request by the remote endpoint timed out.
	EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.
Implementation Notes	The maximum number of vectored I/O structures (struct iovec) in an array per call is sixteen (MSG_IOVLEN).	
See Also	recv(), recvfrom(), recvmsg(), select(), socket(), writev()	

perror()

	System error messages.
Synopsis	<pre>#include <socket.h> #include <uio.h> perror (s) char *s;</pre>
Description	<p>Produces a short error message on the standard error file describing the last error encountered during a call to the socket library for a C program.</p> <p>perror() prints a system-specific error message generated as a result of a failed call to the socket library. The argument string s is printed, followed by a colon, the system error message, and a new line. Most usefully, the argument string is the name of the program that incurred the error. The errno is taken from the external variable errno that is set when errors occur but not cleared when non-erroneous calls are made.</p> <p>To simplify variant formatting of messages, the vector of message strings, sock_errlist is provided; errno is used as an index in this table to get the message string without the new line.</p>
Implementation Notes	This implementation of perror() function front ends the C library perror() function. The list of socket errors is stored in an array of character pointers in the variable sock_errlist. Instead of using sys_nerr variable, the maximum number of error codes defined by the constant ESMAX is used.

recv()

Receive a message on a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
int recv ( s, buf, len, flags )
int s;
char *buf;
int len, flags;
```

Description

Receives incoming data that has been queued for a socket. This function is normally used to receive a reliable, ordered stream of data bytes on a socket of type `SOCK_STREAM`, but can also be used to receive datagrams on a socket of type `SOCK_ASSOC` if an association was formed with a `connect()` function.

The `recv()` function is normally used only on a connected socket (see [connect\(\)](#) for more information), while `recvfrom()` and `recvmsg()` can be used to receive data on a socket whether it is in a connected state or not.

The address of the buffer into which the message is to be received is given by `buf`, and its size is given by `len`. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see [socket\(\)](#) for more information).

If no messages are available at the socket, `recv()` waits for a message to arrive, unless the socket is non-blocking (read [ioctl\(\)](#) for more information) in which case a value of `-1` is returned with the external variable `errno` set to `EWOULDBLOCK`. The `select()` function can be used to determine when more data arrives.

The `flags` argument to `recv()` can be set to `MSG_OOB` (from `<socket.h>`) to receive out-of-band data, but only if out-of-band data is supported by the underlying protocol. Otherwise, `flags` should be set to zero.

Return Value

The `recv()` function returns the number of bytes received if successful. A value of zero is returned if an end-of-file condition exits, indicating no more `recv()` functions should be issued to this socket. Otherwise, the value `-1` is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `recv()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EBADF</code>	An invalid descriptor was specified.
<code>ECONNABORTED</code>	The incoming connection request was aborted by the remote endpoint.
<code>ECONNREFUSED</code>	The remote endpoint refused to continue the connection.
<code>ECONNRESET</code>	The remote endpoint reset the connection request.
<code>EDESTUNREACH</code>	The remote destination is now unreachable.
<code>EFAULT</code>	The buffer passed by the user points to inaccessible memory.
<code>EFAULT</code>	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.
<code>EFAULT</code>	The pointer to the length of the name buffer points to inaccessible storage.
<code>EHOSTUNREACH</code>	The remote host is now unreachable.
<code>EINVAL</code>	The number of bytes in the receive buffer is less than zero.
<code>EINVAL</code>	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
<code>EINVAL</code>	A connectionless socket does not yet have a name bound to it.
<code>EINVAL</code>	A connectionless socket is being used and the length of the name passed by the user is zero.
<code>ENETDOWN</code>	The local network interface is down.
<code>ENOBUFS</code>	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
<code>ENOBUFS</code>	The socket library cannot allocate the necessary buffer space within the API.
<code>ENOTCONN</code>	The socket being used requires a connection before data can be transferred, and no such connection currently exists.

ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes The peek option is not supported for `recv()`, and if `flags` is set to `MSG_PEEK`, an error (`EFAULT`) is generated. The option to receive out-of-band data (`MSG_OOB`) is not supported. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is 16 (`MSG_IOVLEN`).

See Also `recvfrom()`, `recvmsg()`, `read()`, `send()`, `select()`, `getsockopt()`, `socket()`

recvfrom()

Receive a datagram on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int recvfrom ( s, buf, len, flags, from, fromlen )
int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;
```

Description

Receives incoming data that has been queued for a socket. This function normally is used to receive datagrams on a socket of type `SOCK_ASSOC`, but can also be used to receive a reliable, ordered stream of data bytes on a connected socket of type `SOCK_STREAM`.

The `recvfrom()` function normally is used to receive datagrams from a socket, indicated by `s`. If `from` is non-zero, the source address of the datagram is filled in. The `fromlen` parameter is a value-result parameter, initialized to the size of the buffer associated with `from` and modified on return to indicate the actual size of the address stored there.

The address of a buffer into which the datagram is to be received is given by `buf`, and its size is given by `len`. The length of the datagram is returned. If a datagram is too long to fit in the supplied buffer, excess bytes might be discarded depending on the type of socket the datagram is received from (see [socket\(\)](#) for more information).

If no datagrams are available at the socket, `recvfrom()` waits for a datagram to arrive, unless the socket is non-blocking (see [ioctl\(\)](#) for more information), in which case a value of 1 is returned with the external variable `errno` set to `EWOULDBLOCK`. The `select()` function can be used to determine when more data arrives.

The flags argument to `recvfrom()` can be set to `MSG_OOB` (from `<socket.h>`) to receive out-of-band data, but only if out-of-band data is supported by the underlying protocol. Otherwise, flags should be set to zero.

Return Value

The `recvfrom()` function returns the number of bytes received if successful. A value of zero is returned if an end-of-file condition exists, indicating no more `recvfrom()` functions should be issued to this socket. Otherwise, the value -1 is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `recvfrom()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EBADF</code>	An invalid descriptor was specified.
<code>ECONNABORTED</code>	The incoming connection request was aborted by the remote endpoint.
<code>ECONNREFUSED</code>	The remote endpoint refused to continue the connection.
<code>ECONNRESET</code>	The remote endpoint reset the connection request.
<code>EDESTUNREACH</code>	The remote destination is now unreachable.
<code>EFAULT</code>	The buffer passed by the user points to inaccessible memory.
<code>EFAULT</code>	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.
<code>EFAULT</code>	The pointer to the length of the name buffer points to inaccessible storage.
<code>EHOSTUNREACH</code>	The remote host is now unreachable.
<code>EINVAL</code>	The number of bytes in the receive buffer is less than zero.
<code>EINVAL</code>	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
<code>EINVAL</code>	A connectionless socket does not yet have a name bound to it.
<code>EINVAL</code>	A connectionless socket is being used and the length of the name passed by the user is zero.
<code>ENETDOWN</code>	The local network interface is down.

ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
ENOTCONN	The socket being used requires a connection before data may be transferred, and no such connection currently exists.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes The peek option is not supported for `recvfrom()`, and if `flags` is set to `MSG_PEEK`, it is ignored. The option to receive out-of-band data (`MSG_OOB`) is not supported and is also ignored. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is `16(MSG_IOVLEN)`.

See Also `recv()`, `recvmsg()`, `read()`, `send()`, `select()`, `getsockopt()`, `socket()`

recvmsg()

	Receive a message on a socket
Synopsis	<pre>#include <socket.h> #include <uio.h> int recvmsg (s, msg, flags) int s; struct msghdr msg []; int flags;</pre>
Description	<p>Receives incoming data that has been queued for a connected or unconnected socket <i>s</i>. Data is received in <i>scatter</i> mode and placed into noncontiguous buffers.</p> <p>The argument <i>msg</i> is the pointer of a structure, <i>msghdr</i>, used to minimize the number of directly supplied parameters. This structure has this form, as defined in <i><socket.h></i>:</p> <pre>struct msghdr { caddr_t msg_name; int msg_namelen; struct iovec *msg_iov; int msg_iovlen; caddr_t msg_accrights; int msg_accrightslen; };</pre> <p>Here <i>msg_name</i> and <i>msg_namelen</i> specify the source address if the socket is unconnected; <i>msg_name</i> can be given as a null pointer if the source address is not desired or required. The <i>msg_iov</i> and <i>msg_iovlen</i> describe the “scatter” locations, as described in read(). A buffer to receive any access rights sent along with the message is specified in <i>msg_accrights</i>, which has length <i>msg_accrightslen</i>. Access rights are currently limited to file descriptors, with each occupying the size of an <i>int</i>.</p> <p>The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes might be discarded depending on the type of socket the message is received from (see socket() for more information).</p> <p>If no messages are available at the socket, <i>recvmsg()</i> waits for a message to arrive, unless the socket is non-blocking (see ioctl() for more information), in which case a value of 1 is returned with the external variable <i>errno</i> set to <i>EWOULDBLOCK</i>. The <i>select()</i> function can be used to determine when more data arrives.</p> <p>The <i>flags</i> argument to <i>recvmsg()</i> can be set to <i>MSG_OOB</i> (from <i><socket.h></i>) to receive out-of-band data, but only if out-of-band data is supported by the underlying protocol. Otherwise, <i>flags</i> should be set to zero.</p>

Return Value	The <code>recvmsg()</code> function returns the number of bytes received if successful. A value of zero is returned if an end-of-file condition exits, indicating no more <code>recvmsg()</code> functions should be issued to this socket. Otherwise, the value -1 is returned, and the error code stored in the global integer <code>errno</code> indicates the nature of the error.																														
Error Codes	The <code>recvmsg()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code> : <table><tr><td>EBADF</td><td>An invalid descriptor was specified.</td></tr><tr><td>ECONNABORTED</td><td>The incoming connection request was aborted by the remote endpoint.</td></tr><tr><td>ECONNREFUSED</td><td>The remote endpoint refused to continue the connection.</td></tr><tr><td>ECONNRESET</td><td>The remote endpoint reset the connection request.</td></tr><tr><td>EDESTUNREACH</td><td>Remote destination is now unreachable.</td></tr><tr><td>EFAULT</td><td>The buffer passed by the user points to inaccessible memory.</td></tr><tr><td>EFAULT</td><td>The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.</td></tr><tr><td>EFAULT</td><td>The pointer to the length of the name buffer points to inaccessible storage.</td></tr><tr><td>EFAULT</td><td>The <code>msg_hdr</code> pointer points to inaccessible memory.</td></tr><tr><td>EFAULT</td><td>The <code>iovec</code> structure pointer within the <code>msg_hdr</code> structure points to inaccessible memory.</td></tr><tr><td>EHOSTUNREACH</td><td>Remote host is now unreachable.</td></tr><tr><td>EINVAL</td><td>The number of bytes in the receive buffer is less than zero.</td></tr><tr><td>EINVAL</td><td>The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.</td></tr><tr><td>EINVAL</td><td>A connectionless socket does not yet have a name bound to it.</td></tr><tr><td>EINVAL</td><td>A connectionless socket is being used and the length of the name passed by the user is zero.</td></tr></table>	EBADF	An invalid descriptor was specified.	ECONNABORTED	The incoming connection request was aborted by the remote endpoint.	ECONNREFUSED	The remote endpoint refused to continue the connection.	ECONNRESET	The remote endpoint reset the connection request.	EDESTUNREACH	Remote destination is now unreachable.	EFAULT	The buffer passed by the user points to inaccessible memory.	EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.	EFAULT	The pointer to the length of the name buffer points to inaccessible storage.	EFAULT	The <code>msg_hdr</code> pointer points to inaccessible memory.	EFAULT	The <code>iovec</code> structure pointer within the <code>msg_hdr</code> structure points to inaccessible memory.	EHOSTUNREACH	Remote host is now unreachable.	EINVAL	The number of bytes in the receive buffer is less than zero.	EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.	EINVAL	A connectionless socket does not yet have a name bound to it.	EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero.
EBADF	An invalid descriptor was specified.																														
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.																														
ECONNREFUSED	The remote endpoint refused to continue the connection.																														
ECONNRESET	The remote endpoint reset the connection request.																														
EDESTUNREACH	Remote destination is now unreachable.																														
EFAULT	The buffer passed by the user points to inaccessible memory.																														
EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.																														
EFAULT	The pointer to the length of the name buffer points to inaccessible storage.																														
EFAULT	The <code>msg_hdr</code> pointer points to inaccessible memory.																														
EFAULT	The <code>iovec</code> structure pointer within the <code>msg_hdr</code> structure points to inaccessible memory.																														
EHOSTUNREACH	Remote host is now unreachable.																														
EINVAL	The number of bytes in the receive buffer is less than zero.																														
EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.																														
EINVAL	A connectionless socket does not yet have a name bound to it.																														
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero.																														

EINVAL	The number of iovec structures specified by the msghdr is less than or equal to zero.
EMSGSIZE	The number of iovec structures specified by the msghdr structure is greater than 16.
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.
Implementation Notes	The peek option is not supported for <code>recvmsg()</code> , and if <code>flags</code> is set to <code>MSG_PEEK</code> , it is ignored. The option to receive out-of-band data (<code>MSG_OOB</code>) is not supported and is ignored. The maximum number of vectored I/O structures (<code>struct iovec</code>) in an array per call is 16 (<code>MSG_IOVLEN</code>).
See Also	<code>recv()</code> , <code>recvfrom()</code> , <code>read()</code> , <code>send()</code> , <code>select()</code> , <code>getsockopt()</code> , <code>socket()</code>

select()

Synchronous I/O multiplexing.

Synopsis

```
#include <socket.h>

#include <uio.h>
int select ( nfds, readfds, writefds, exceptfds, timeout )
int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
FD_SET ( fd, &fdset )
FD_CLR ( fd, &fdset )
FD_ISSET ( fd, &fdset )
FD_ZERO ( &fdset )
int fd;
fd_set fdset;
```

Description

Synchronizes processing of several sockets operating in non-blocking mode. Sockets that are ready for reading, ready for writing, or have a pending exceptional condition can be selected. If no sockets are ready for processing, the `select()` function can block indefinitely or wait for a specified period of time (which may be zero) and then return.

`select()` examines the I/O descriptor sets whose addresses are passed in `readfds`, `writefds`, and `exceptfds` to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first `nfds` descriptors are checked in each set (that is, the descriptors from zero through `nfds-1` in the descriptor sets are examined). On return, `select()` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned.

The descriptor sets are stored as bit fields in arrays of integers. These macros are provided for manipulating such descriptor sets:

<code>FD_ZERO (&fdset)</code>	Initializes a descriptor set <code>fdset</code> to the null set.
<code>FD_SET (fd, &fdset)</code>	Includes a particular descriptor <code>fd</code> in <code>fdset</code> .
<code>FD_CLR (fd, &fdset)</code>	Removes <code>fd</code> from <code>fdset</code> .
<code>FD_ISSET (fd, &fdset)</code>	Is non-zero if <code>fd</code> is a member of <code>fdset</code> , zero otherwise.

The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`, which normally is at least equal to the maximum number of descriptors supported by the system.

If `timeout` is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If `timeout` is a zero pointer, the `select` blocks indefinitely. To affect a poll, the `timeout` argument should be non-zero, pointing to a zero-valued `timeval` structure.

Any of readfds, writefds, and exceptfds can be given as zero pointers if no descriptors are of interest.

Sockets use MVS STIMERM services. Socket applications may use up to fifteen STIMERM calls per task control block (TCB). STIMER CANCEL ID=ALL must not be used by socket applications.

Note: For release 3.1 and higher, note that the select() and mvselect() functions have changed slightly for SAS/C users. SAS/C allows signals to be raised. select() may complete if the signal is raised without the conditions on the select() parameters having been met. This new feature allows you to use signal functionality, issue a blocking receive, and do a send from a signal handler.

Return Value	If successful, select() returns the number of ready descriptors that are contained in the descriptor sets, or zero if the time limit expires. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error. If select() returns with an error, including one due to an interrupted call, the descriptor sets are unmodified.
Error Codes	The select() function returns these error codes (from <errno.h>) in the global integer errno: EBADF One of the descriptor sets specified an invalid descriptor. EINVAL The specified time limit is invalid. One of its components is negative or too large. EINVAL The count of socket descriptors to check is less than or equal to zero. EINVAL No valid socket descriptors were referenced by the three sets of masks passed by the caller.
Implementation Notes	The implementation of select() provided with the API supports only descriptors associated with sockets.
See Also	accept(), connect(), read(), write(), recv(), send(), recvfrom(), sendto(), recvmsg(), sendmsg(), mvselect()

send()

Send a message on a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
int send ( s, msg, len, flags )
int s;
char *msg;
int len, flags;
```

Description

Sends outgoing data on a connected socket *s*. This function is normally used to send a reliable, ordered stream of data bytes on a socket of type `SOCK_STREAM`, but can also be used to send datagrams on a socket of type `SOCK_ASSOC`, if an association has been formed with a `connect()` function.

The `send()` function normally is used only on a connected socket (see [connect\(\)](#) for more information), while `sendto()` and `sendmsg()` can be used to send data on a socket whether it is connected or not.

The location of the message is given by *msg*, and its size is given by *len*. The number of bytes sent is returned. If the message is too long to pass automatically through the underlying protocol, the error `EMSGSIZE` is returned, and the message is not transmitted.

If no buffer space is available at the socket to hold the message to be transmitted, `send()` normally blocks. However, if the socket has been placed in non-blocking I/O mode (see [ioctl\(\)](#) for more information), a value of `-1` is returned with the external variable `errno` set to `EMSGSIZE`, and the message is not transmitted. The `select()` function can be used to determine when it is possible to send more data.

The *flags* argument to `send()` may be set to `MSG_OOB` (from `<socket.h>`) to send out-of-band data, if the underlying protocol supports this notion. Otherwise, *flags* should be set to zero.

Return Value

If successful, `send()` returns the number of bytes sent. Otherwise, the value `-1` is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The send() function returns these error codes (from <errno.h>) in the global integer errno:

EADDRNOTAVAIL	A connectionless socket had a send request issued to it, but the user passed a name of the remote endpoint that was invalid. Either the remote port is zero or the remote address is INADDR_ANY.
EAFNOSUPPORT	The name of the remote endpoint to send the data to specified a domain other than AF_INET.
EBADF	An invalid descriptor was specified.
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTADDRREQ	A connectionless socket is being used and no name of the remote endpoint has been passed by the user.
EDESTUNREACH	Remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the name of the remote endpoint passed by the user points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes to transmit is less than or equal to zero.
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero.
EINVAL	A send request was issued to a socket that is operating in connectionless mode, but the user did not pass a name of the remote endpoint to which to send the data.
EISCONN	A socket associated with a remote endpoint has been issued a send request in which the user specified a remote endpoint name.
EMSGSIZE	The socket requires that the message be sent automatically, and the size of the message to be sent made this impossible.

ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but it can be caused by transient congestion.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
EOPNOTSUPP	The user tried to send urgent data (MSG_OOB) on a socket that does not support this concept.
EPIPE	An attempt was made to send to a socket that is not connected to or associated with a peer socket.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes The `send()` function does not support the `MSG_DONTROUTE` option and should not be set in flags. The option `MSG_MORE` lets the user program inform sockets of the fact that more data is about to be written to the socket. This may be used by the transport provider to influence its packetizing algorithm. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is 16 (`MSG_IOVLEN`).

See Also `recv()`, `sendto()`, `sendmsg()`, `select()`, `getsockopt()`, `socket()`, `write()`

sendmsg()

Send a message on a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
int sendmsg ( s, msg, flags )
int s;
struct msghdr msg [ ];
int flags;
```

Description

Sends outgoing data on a connected or unconnected socket *s*. Data is sent in gather mode from a list of noncontiguous buffers.

The argument *msg* is a pointer to a structure, *msghdr*, used to minimize the number of directly supplied parameters. This structure has this form, as defined in *<socket.h>*:

```
struct msghdr
{
    caddr_t      msg_name;
    int          msg_namelen;
    struct iovec *msg_iov;
    int          msg_iovlen;
    caddr_t      msg_accrights;
    int          msg_accrightslen;
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected. *msg_name* can be given as a null pointer if the destination address is not required. The *msg_iov* and *msg_iovlen* describe the gather locations, as described in [writev\(\)](#). A buffer containing any access rights to send with the message is specified in *msg_accrights*, which has length *msg_accrightslen*. Access rights are currently limited to file descriptors, each occupying the size of an int.

The number of bytes sent is returned. If the message is too long to pass automatically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

If no message space is available at the socket to hold the message to be transmitted, *sendmsg()* normally blocks. However, if the socket has been placed in non-blocking I/O mode (see [ioctl\(\)](#) for more information), a value of -1 is returned with the external variable *errno* set to EMSGSIZE, and the message is not transmitted. The *select()* function can be used to determine when it is possible to send more data.

The *flags* argument to *sendmsg()* can be set to MSG_OOB (from *<socket.h>*) to send out-of-band data if the underlying protocol supports this notion. Otherwise, *flags* should be set to zero.

Return Value	If successful, <code>sendmsg()</code> returns the number of bytes sent. Otherwise, the value -1 is returned, and the error code stored in the global integer <code>errno</code> indicates the nature of the error.	
Error Codes	The <code>sendmsg()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code> :	
	<code>EADDRNOTAVAIL</code>	A connectionless socket had a send request issued to it, but the user passed a name of the remote endpoint that was invalid. Either the remote port is zero or the remote address is <code>INADDR_ANY</code> .
	<code>EAFNOSUPPORT</code>	The name of the remote endpoint to send the data to specified a domain other than <code>AF_INET</code> .
	<code>EBADF</code>	An invalid descriptor was specified.
	<code>ECONNABORTED</code>	The incoming connection request was aborted by the remote endpoint.
	<code>ECONNREFUSED</code>	The remote endpoint refused to continue the connection.
	<code>ECONNRESET</code>	The remote endpoint reset the connection request.
	<code>EDESTADDRREQ</code>	A connectionless socket is being used and no name of the remote endpoint has been passed by the user.
	<code>EDESTUNREACH</code>	Remote destination is now unreachable.
	<code>EFAULT</code>	The buffer passed by the user points to inaccessible memory.
	<code>EFAULT</code>	The pointer to the name of the remote endpoint passed by the user points to inaccessible memory.
	<code>EFAULT</code>	The pointer to the <code>msg_hdr</code> structure points to inaccessible memory.
	<code>EFAULT</code>	The <code>msg_hdr</code> has an <code>iovec</code> pointer that points to inaccessible memory.
	<code>EHOSTUNREACH</code>	Remote host is now unreachable.
	<code>EINVAL</code>	The number of bytes to transmit is less than or equal to zero.
	<code>EINVAL</code>	A connectionless socket does not yet have a name bound to it.
	<code>EINVAL</code>	A connectionless socket is being used and the length of the name passed by the user is zero.

EINVAL	A send request was issued to a socket that is operating in connectionless mode, but the user did not pass a name of the remote endpoint to which to send the data.
EINVAL	The msghdr structure specifies an array of less than or equal to zero iovec elements.
EISCONN	A socket that is associated with a remote endpoint has been issued a send request in which the user specified a remote endpoint name.
EMSGSIZE	The socket requires that the message be sent automatically, and the size of the message to be sent made this impossible.
EMSGSIZE	The msghdr structure specifies an array of iovec elements of less than or equal to zero.
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but can be caused by transient congestion.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
EOPNOTSUPP	The user tried to send urgent data (MSG_OOB) on a socket that does not support this concept.
EPIPE	An attempt was made to send to a socket that is not connected to or associated with a peer socket.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWouldBlock	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The `sendmsg()` function does not support the `MSG_DONTROUTE` option and should not be set in flags. The option `MSG_MORE` lets the user program inform sockets of the fact that there is more data about to be written to the socket. This can be used by the transport provider to influence its packetizing algorithm. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is 16 (`MSG_IOVLEN`).

See Also

`recv()`, `sendto()`, `sendmsg()`, `select()`, `getsockopt()`, `socket()`, `write()`

sendto()

Send a datagram on a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
int sendto ( s, msg, len, flags, to, tolen )
int s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;
```

Description

Sends outgoing data on a connected or unconnected socket. This function normally is used to send datagrams on a socket of type `SOCK_DGRAM`, but can also be used to send a reliable, ordered stream of data bytes on a connected socket of type `SOCK_STREAM`.

The `sendto()` function normally is used to transmit a datagram from a socket that is unconnected. The socket is indicated by `s`. The destination address is given by `to`, and its length is given by `tolen`. If the socket is connected or associated with a destination (see [connect\(\)](#) for more information), `to` and `tolen` can be set to zero.

The location of the datagram is given by `msg`, and its size is given by `len`. The number of bytes sent is returned. If the datagram is too long to pass automatically through the underlying protocol, the error `EMSGSIZE` is returned, and the datagram is not transmitted.

If no buffer space is available at the socket to hold the datagram to be transmitted, `sendto()` normally blocks. However, if the socket has been placed in non-blocking I/O mode (see [ioctl\(\)](#) for more information), a value of `-1` is returned with the external variable `errno` set to `EMSGSIZE`, and the datagram is not transmitted. The `select()` function can be used to determine when it is possible to send more data.

The `flags` argument to `sendto()` can be set to `MSG_OOB` (from `<socket.h>`) to send out-of-band data, if the underlying protocol supports this notion. Otherwise, `flags` should be set to zero.

Return Value

If successful, `sendto()` returns the number of bytes sent. Otherwise, the value `-1` is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `sendto()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EADDRNOTAVAIL</code>	A connectionless socket had a send request issued to it, but the user passed a name of the remote endpoint that was invalid. Either the remote port is zero or the remote address is <code>INADDR_ANY</code> .
<code>EAFNOSUPPORT</code>	The name of the remote endpoint to send the data to specified a domain other than <code>AF_INET</code> .
<code>EBADF</code>	An invalid descriptor was specified.
<code>ECONNABORTED</code>	The incoming connection request was aborted by the remote endpoint.
<code>ECONNREFUSED</code>	The remote endpoint refused to continue the connection.
<code>ECONNRESET</code>	The remote endpoint reset the connection request.
<code>EDESTADDRREQ</code>	A connectionless socket is being used and no name of the remote endpoint has been passed by the user.
<code>EDESTUNREACH</code>	The remote destination is now unreachable.
<code>EFAULT</code>	The buffer passed by the user points to inaccessible memory.
<code>EFAULT</code>	The pointer to the name of the remote endpoint passed by the user points to inaccessible memory.
<code>EHOSTUNREACH</code>	The remote host is not unreachable.
<code>EINVAL</code>	The number of bytes to transmit is less than or equal to zero.
<code>EINVAL</code>	A connectionless socket does not yet have a name bound to it.
<code>EINVAL</code>	A connectionless socket is being used and the length of the name passed by the user is zero.
<code>EINVAL</code>	A send request was issued to a socket that is operating in connectionless mode, but the user did not pass a name of the remote endpoint to which to send the data.
<code>EISCONN</code>	A socket that is associated with a remote endpoint has been issued a send request in which the user specified a remote endpoint name.
<code>EMSGSIZE</code>	The socket requires that the message be sent automatically, and the size of the message to be sent made this impossible.
<code>ENETDOWN</code>	The local network interface is down.

ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but can be caused by transient congestion.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
EOPNOTSUPP	The user tried to send urgent data (MSG_OOB) on a socket that does not support this concept.
EPIPE	An attempt was made to send to a socket that is not connected to or associated with a peer socket.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation
Notes

The `sendto()` function does not support the `MSG_DONTROUTE` option and should not be set in flags. The option `MSG_MORE` lets the user program inform sockets of the fact that there is more data about to be written to the socket. This can be used by the transport provider to influence its packetizing algorithm. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is 16 (`MSG_IOVLEN`).

See Also

`recv()`, `send()`, `sendmsg()`, `select()`, `getsockopt()`, `socket()`, `write()`

setlogmask()

Set log priority mask

Synopsis

```
#include <syslog.h>
int setlogmask ( maskpri )
int maskpri;
```

Description

The `setlogmask()` function sets the log priority mask to `maskpri` and returns the previous mask. Calls to `syslog()` with a priority not set in `maskpri` are rejected. The mask for an individual priority `pri` is calculated by the macro `LOG_MASK (pri)`; the mask for all priorities up to and including `toppri` is given by the macro `LOG_UPTO (toppri)`. The default allows all priorities to be logged.

See Also

`closelog()`, `openlog()`, `syslog()`, `vsyslog()`

setsockopt()

Set options on a socket.

Synopsis

```
#include <socket.h>
#include <uio.h>
#include <inet.h>
#include <tcp.h>
int setsockopt ( s, level, optname, optval, optlen )
int s, level, optname;
char *optval;
int optlen;
```

Description

Manipulates options associated with a socket. Options always exist at the socket level and can also exist at layers within the underlying protocols. Options are retrieved with the `getsockopt()` function. Options can exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as `SOL_SOCKET`. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied.

Example

To indicate that a TCP protocol option is to be changed, level should be set to the protocol number of TCP. Read [getprotobyname\(\)](#) for more information.

The parameters `optval` and `optlen` identify a buffer that contains the option value. The `optlen` parameter is the length of the option value in bytes.

The `optname` parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file `<socket.h>` contains definitions for socket level options. Options at other protocol levels vary in format and name; consult the appropriate appendix for additional information.

Most socket-level options require an `int` parameter for `optval`. For boolean options, a non-zero value indicates the option is to be enabled, and a zero value indicates the option is to be disabled. `SO_LINGER` uses a struct `linger` parameter, defined in `<socket.h>`, that specifies the desired state of the option and the linger interval.

The following table lists options recognized at the socket level. Except as noted, each can be set with `setsockopt()` and examined with `getsockopt()`:

Option	Definition
SO_BROADCAST	Requests permission to send broadcast datagrams on the socket.
SO_DEBUG	Enables debugging in the socket modules. This option is implemented slightly differently than UNIX in that on UNIX it allows for debugging at the underlying protocol modules.
SO_DONTROUTE	Indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_KEEPALIVE	Enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via an <code>errno</code> <code>ETIMEDOUT</code> .
SO_LINGER	Controls the action taken when unsent messages are queued on socket and a <code>close()</code> is performed. If the socket promises reliable delivery of data and <code>SO_LINGER</code> is set to a value other than zero, the system blocks the process on the <code>close()</code> attempt until it is able to transmit the data or until the number of seconds specified by the <code>SO_LINGER</code> option expire or until the connection has timed out. If <code>SO_LINGER</code> is set and the interval is zero, the socket is closed once the system has scheduled that an orderly release be performed.
SO_OOINLINE	This option is not currently supported by this release of the socket library. It is implemented on UNIX, and its purpose is to request that out-of-band data be placed in the normal data input queue as received by protocols that support out-of-band data; it is then accessible with <code>recv()</code> or <code>read()</code> functions without the <code>MSG_OOB</code> flag.
SO_RCVBUF	Adjusts the normal buffer size allocated for input. The buffer size can be increased for high-volume connections to improve throughput or can be decreased to limit the possible backlog of incoming data. The system places an absolute limit on this value. This implementation of sockets provides for this option for backward compatibility, but also allows for buffer options that are more specific to the underlying API and therefore provide a better method of controlling a socket's buffering characteristics. These options are <code>SO_RCVBYTCNT</code> and <code>SO_RCVREQCNT</code> . All buffering options can be set only once by the socket user and must be done before any data is sent or received on the socket.
SO_RCVBYTCNT	Adjusts the number of bytes allocated to the receive circular buffer for a socket. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the receive byte count of the circular buffer is changed, it must be done prior to sending or receiving any data on the socket.
SO_RCVLOWAT	Adjusts the size of the receive low water mark.

Option	Definition
SO_RCVREQCNT	Adjusts the number of receive requests that can be active within the socket library at a given time. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the receive request count is changed, it must be done prior to sending or receiving any data on the socket.
SO_READFRAG	This option lets a user of a datagram type socket that preserves message boundaries read a datagram a piece at a time. Traditionally, with UNIX sockets, if a user issues a read request for 100 bytes and the datagram being read consists of 120 bytes, the socket returns the first 100 bytes to the caller and then flushes the remaining 20 bytes. This default method of operation by this implementation of sockets can be overridden by using this option. This option does not allow for the parsing of pieces of a single datagram into a single user buffer. When this option is used, the user must determine the boundaries of datagrams. This option is specific to this implementation and is not portable to other socket implementations.
SO_REUSEADDR	Indicates that the rules used in validating addresses supplied in a bind() function should allow reuse of local addresses.
SO_SENDALL	This option guarantees that any form of send request (send(), sendto(), sendmsg(), write(), or writev()) that is done in a blocking mode transmits all the data specified by the user. Traditionally, BSD sockets send as many bytes as the current buffering allocation allowed for and then return to the user with a count of the actual number of bytes transmitted. If the user requested that a write of 200 bytes be done but there currently was only buffering space for 150 bytes, the socket queues 150 bytes for transmission and returns a count of 150 to the caller to indicate that 50 bytes could not be transmitted due to lack of buffer space. This implementation of sockets acts identically to a UNIX socket under the same scenario under the default setting of the socket options. However, if this option is turned ON in Unicenter TCPaccess sockets, the socket blocks the user until all of the data has been queued for transmission or some type of error occurs. This option is specific to this implementation.
SO_SNDBUF	Adjusts the normal buffer size allocated for output. The buffer size can be increased for high-volume connections to improve throughput or can be decreased to limit the possible backlog of outgoing data. The system places an absolute limit on this value. This implementation of sockets provides this option for backward compatibility. It also allows for buffer options that are more specific to the underlying API and therefore provides a better method of controlling a socket's buffering characteristics. These options are SO_SNDBYTCNT and SO_SNDREQCNT. All buffering options can be set only once by the socket user and must be done before any data is sent or received on the socket.

Option	Definition
SO_SNDBYTCNT	Adjusts the number of bytes allocated to the send circular buffer for a socket. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the send byte count of the circular buffer is changed, it must be done prior to sending or receiving any data on the socket.
SO_SNDLOWAT	Adjusts the size of the send low water mark.
SO_SNDREQCNT	Adjusts the number of send requests that can be active within the socket library at a given time. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the send request count is changed, it must be done prior to sending or receiving any data on the socket.
SO_USELOOPBACK	Requests that the loopback interface is used rather than a real physical interface.

The following table lists TCP level options:

Option	Definition
TCP_NODELAY	Ensures that TCP type sockets (SOCK_STREAM) send the data as soon as possible and do not wait for more data or a given amount of time to enhance the packetizing algorithm. This option is similar to the BSD UNIX socket option.
TCP_MAXSEG	This option is not supported by this implementation of sockets at the present release. On UNIX, this option lets the user of a SOCK_STREAM socket declare the value of the maximum segment size for TCP to use when negotiating this value with its remote endpoint.

The following option is at the UDP level.

UDP_CHECKSUM Sets whether UDP checksum computation is to be performed.

The following table lists options at the IP level

Option	Definition
IP_HDRINCL	Specifies that the application include the IP header in data for the SEND option. Applicable for RAW sockets only
IP_OPTIONS	Sets specific options in the IP header
IP_TOS	Sets the type-of-service field in IP header of outgoing packets.
IP_TTL	Sets the time-to-live field in IP header of outgoing packets.

Return Value	If <code>setsockopt()</code> is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer <code>errno</code> indicates the nature of the error.
Error Codes	<p>The <code>setsockopt()</code> function returns these error codes (from <code><errno.h></code>) in the global integer <code>errno</code>:</p> <p><code>EBADF</code> The <code>s</code> argument is not a valid descriptor.</p> <p><code>EFAULT</code> The pointer to the value buffer points to inaccessible memory.</p> <p><code>EFAULT</code> The pointer to the value buffer length points to inaccessible memory.</p> <p><code>EINVAL</code> The size of the value buffer does not equal the size of the option. Most options require an integer length buffer, or in the case of <code>SO_LINGER</code> the buffer must be the size of the <code>linger</code> structure.</p> <p><code>EINVAL</code> The size of the option buffer is greater than the maximum allowed by the API.</p> <p><code>EINVAL</code> The option is not supported at the level requested.</p> <p><code>EINVAL</code> No options can be read from the protocol layers.</p>
Implementation Notes	<p>The following options are recognized at the socket level on BSD UNIX systems and for OpenEdition Converged sockets, but are not supported by the API. If any of these options are referenced, an error is generated:</p> <p><code>SO_RCVLOWAT</code> Sets the receive low water buffering mark.</p> <p><code>SO_RCVTIMEO</code> Sets the receive timeout value. This option is not currently implemented in UNIX.</p> <p><code>SO_SNDLOWAT</code> Sets the send low water buffering mark.</p> <p><code>SO_SNDTIMEO</code> Sets the send timeout value. This option is not currently implemented in UNIX.</p> <p>The effect of setting the supported socket level options can differ from that which occurs in a UNIX environment.</p>
Example	The <code>SO_DEBUG</code> option enables the API debugging facilities, but the output produced by those facilities can differ from that produced on a UNIX system.
See Also	<code>getsockopt()</code> , <code>ioctl()</code> , <code>socket()</code>

shutdown()

Shut down part of a full-duplex connection

Synopsis

```
#include <socket.h>
#include <uio.h>
int shutdown ( s, how )
int s, how;
```

Description

Gracefully shuts down a socket. The input path can be shut down while continuing to send data, the output path can be shut down while continuing to receive data, or the socket can be shut down in both directions at once. Data queued for transmission is not lost.

The shutdown() function causes all or part of a full-duplex connection on the socket associated with s to be shut down.

If how is zero, further receives are disallowed. If how is one, further sends are disallowed. If how is 2, further sends and receives are disallowed.

Return Value

If shutdown() is successful, a value of zero is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The shutdown() function returns these error codes (from <errno.h>) in the global integer errno:

EBADF	The s argument is not a valid descriptor.
ECONNABORTED	The connection was aborted by a local action of the API.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
ETIMEDOUT	The connection timed out.

See Also

connect(), socket()

socket()

	Create an endpoint for communication.								
Synopsis	<pre>#include <socket.h> #include <uio.h> int socket (domain, type, protocol) int domain, type, protocol;</pre>								
Description	<p>Creates an endpoint in a communications domain. The endpoint is called a socket. When the socket is created, a protocol is selected and a descriptor is returned to represent the socket. The socket descriptor is used in all subsequent functions referencing the socket. Only sockets in the Internet domain using TCP or UDP protocol are supported by this implementation.</p> <p>The domain argument specifies a communications domain within which communication takes place; this selects the protocol family to use. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file <socket.h>.</p> <p>This is the only protocol family currently recognized by this implementation:</p> <pre>PF_INET (Internet protocols)</pre> <p>The socket has the indicated type, which specifies the semantics of communication. The following are currently defined types:</p> <table><tr><td>SOCK_STREAM</td><td>Provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism can be supported.</td></tr><tr><td>SOCK_DGRAM</td><td>Supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).</td></tr><tr><td>SOCK_ASSOC</td><td>Supports datagrams (associations, unreliable messages of a fixed (typically small) maximum length).</td></tr><tr><td>SOCK_RAW</td><td>Supports datagrams (connectionless, unreliable messages of a fixed maximum length) with the complete IP header included in each datagram.</td></tr></table>	SOCK_STREAM	Provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism can be supported.	SOCK_DGRAM	Supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).	SOCK_ASSOC	Supports datagrams (associations, unreliable messages of a fixed (typically small) maximum length).	SOCK_RAW	Supports datagrams (connectionless, unreliable messages of a fixed maximum length) with the complete IP header included in each datagram.
SOCK_STREAM	Provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism can be supported.								
SOCK_DGRAM	Supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).								
SOCK_ASSOC	Supports datagrams (associations, unreliable messages of a fixed (typically small) maximum length).								
SOCK_RAW	Supports datagrams (connectionless, unreliable messages of a fixed maximum length) with the complete IP header included in each datagram.								

Socket Library Include Files

This chapter lists the include files used with the API socket library. These files define structures, macros, and constants referenced by the socket library and should be included during the compilation of any application programs that call socket library functions.

Include File Summary

Socket library include files are installed on the local system as members of a partitioned data set. Include this data set in the SYSLIB DD concatenation when application programs are compiled. The data set name is determined during installation (default name T01TCP.H unless changed by the local system programmer).

The following table summarizes the contents of each include file:

INCLUDE FILE	PDS MEMBER	STRUCTURE DECLARATIONS	MACRO DEFINITIONS	OTHER DEFINITIONS
acs.h	ACS			
cdefs.h	CDEFS		_CONCAT _PA _STRING	ANSI C Keywords
errno.h	ERRNO		GET_ERRNO SET_ERRNO	Socket errno values
if.h	IF	ifnet ifaddr ifreq ifconf	IF_QFULL IF_DROP IF_ENQUEUE IF_PREPEND IF_DEQUEUE	
if_arp	IF#ARP	arp_hdr arpreq		

writev()

INCLUDE FILE	PDS MEMBER	STRUCTURE DECLARATIONS	MACRO DEFINITIONS	OTHER DEFINITIONS
if_ether	IF#ETHER	ether_header ether_arp	ETHER_MAP_IP_MULTICAST	
in.h	IN	in_addr sockaddr_in ip_mreq	IN_CLASSA IN_CLASSB IN_CLASSC IN_CLASSD IN_MULTICAST IN_EXPERIMENTAL IN_BADCLASS htonl htons ntohl ntohs	Protocol defines Well-known port numbers IP options Important IP addresses Function prototypes
inet.h	INET			Includes in.h
iocom.h	IOCCOM		IOC_VOID IOC_OUT IOC_IN IOC_INOUT IOC_DIRMASK	
ioctl.h	IOCTL		SIOCADDRT SIOCDELRT OSIOCGIFADDR SIOCGIFADDR OSIOCGIFDSTADDR SIOCGIFDSTADDR SIOCGIFFLAGS OSIOCGIFBRDADDR SIOCGIFBRDADDR SIOCGIFCONF OSIOCGIFNETMASK SIOCGIFNETMASK SIOCGIFMETRIC SIOCGIFNUM SIOCGIFHWADDR SIOCGIFMTU	

INCLUDE FILE	PDS MEMBER	STRUCTURE DECLARATIONS	MACRO DEFINITIONS	OTHER DEFINITIONS
ip.h	IP	ip ip_timestamp		Type of service values Precedence values IP options values Security values Internet implementation values
netdb.h	NETDB	hostent netent servent protoent		Error return codes Function prototypes
param.h	PARAM		setbit clrbit isset isclr howmany roundup powerof2 MIN MAX	
serrno.h	SERRNO			includes errno.h
sockcfg.h	SOCKCFG	cfuns s0skcfg	SET_CNFG_ERR SET_APICNFG_ERR	Configuration parameters Configuration error codes
socket.h	SOCKET	linger sockaddr sockproto msghdr clientid sstat	bcopy bzero bcmp	Socket types Socket options Address families Protocol families
sockio.h	SOCKIO			includes ioctl.h

writev()

INCLUDE FILE	PDS MEMBER	STRUCTURE DECLARATIONS	MACRO DEFINITIONS	OTHER DEFINITIONS
sockvar.h	SOCKVAR	arb sb tpi sdebug errtbl socket	spending sstate SOISREADABLE TPLGEN SAVE_ERR SAVE_EXIT SAVE_CMPL_EXIT	
syslog.h	SYSLOG			Priority values/names Option flag values Facility names
sys_time.h	SYSTIME	timeval timezone itemerval	timerisset timercmp timerclear	
tcp.h	TCP			TCP_NODELAY TCP_MAXSEG TCP_LINGERTIME
time.h	TIME	tm	difftime	Time values
types.h	TYPES		FD_SET FD_CLR FD_ISSET FD_ZERO	standard C type defines
uio.h	UIO	iovec uio		
unistd.h	UNISTD			STDIN_FILENO STDOUT_FILENO STDERR_FILENO

DNR Directory Services

This chapter describes the C library directory services function `dirsrv()` and both C and assembler language definitions for DPL/dpl, the Directory Services Parameter List.

It includes these sections:

- [The `dirsrv\(\)` Function](#) – Describes the basic C library directory services function `dirsrv()`, which is used to request a service from the Domain Name Resolver (DNR)
- [Directory Services Parameter List \(DPL\)](#) – Includes the assembler language (DPL) and the C language (`dpl`) definitions for the Directory Services Parameter List (DPL).

The `dirsrv()` Function

The `dirsrv()` function is used to request a service from the Domain Name Resolver (DNR). The DNR subsystem maintains a data base of information locally and throughout the network. The `dirsrv()` function provides a general-purpose interface to this information.

Format Description

The following is the format description for this function:

```
#include <ds.h>

int      dirsrv ( dplp )
struct dpl      *dplp;
```

Description

Performs the request specified by the format of the DPL passed to it. In the processing of this request, the DNR subsystem is called to perform the actual request. Any errors detected by the DNR are returned to the caller as the return code (error) of this function. In addition to the general return code, the DPL return code field can be set to a value to more clearly distinguish the reason for the error. A return code of zero signifies proper completion of the request. Along with the normal completion, the DPL return code field can provide a conditional completion code. All completion codes are documented in the section covering the DNR.

Completion On successful completion, `dirsrv()` returns a zero. Otherwise it returns the general completion code of the `DIRSRV` macro of the DNR subsystem. The user should refer to the documentation on the `DIRSRV` macro for more information on error codes it can return.

Directory Services Parameter List (DPL)

This section includes the assembler language and C language definition for the Directory Services Parameter List (DPL).

By convention, the assembler language definition refers to the Directory Services Parameter List as `DPL`; the C language definition refers to it as `dpl`.

Assembler Language Definition

This section contains the Assembler Language Definition for `DPL`, the Directory Services Parameter List.

DPL	DSECT	DIRECTORY SERVICES PARAMETER LIST
DPLIDENT DS	X	CONTROL BLOCK ID
DPLIDSTD EQU	237	STANDARD FORMAT ID
DPLFNCCD DS	X	FUNCTION CODE
DFHSTBYN EQU	1	GET HOST BY NAME
DFHSTBYV EQU	2	GET HOST BY VALUE
DFHSTBYA EQU	3	GET HOST BY ALIAS
DFNETBYN EQU	4	GET NETWORK BY NAME
DFNETBYV EQU	5	GET NETWORK BY VALUE
DFSRVBYN EQU	6	GET SERVICE BY NAME
DFSRVBYV EQU	7	GET SERVICE BY VALUE
DFPROBYN EQU	8	GET PROTOCOL BY NAME
DFPROBYV EQU	9	GET PROTOCOL BY VALUE
DFHSTSRV EQU	10	GET HOST SERVICES (BY NAME)
DFHSTINF EQU	11	GET HOST INFORMATION (BY NAME)
DFROUTE EQU	12	GET MAIL ROUTE
DFRPCBYN EQU	13	GET RPC BY NAME
DFRPCBYV EQU	14	GET RPC BY VALUE
DFMAX EQU	DFRPCBYV	MAXIMUM FUNCTION CODE
DPLACTIV DS	X	SEMAPHORE (DPL ACTIVE)
DPLFLAGS DS	X	FLAG BYTE
DPLFCMPL EQU	B'10000000'	DPL COMPLETED
DPLFCERR EQU	B'01000000'	COMPLETED WITH ERROR
DPLFXECB EQU	B'00100000'	DPLECBXR IS EXTERNAL ECB
DPLFEXIT EQU	B'00010000'	DPLECBXR IS EXIT ROUTINE
DPLF31B EQU	B'00001000'	REQUEST ISSUED WITH AMODE=31
* EQU	B'00000100'	RESERVED
* EQU	B'00000010'	RESERVED
* EQU	B'00000001'	RESERVED
DPLSYSID DS	CL4	MVS SUBSYSTEM ID
DPLECBXR DS	A	IECB/XECB/EXIT
ORG	*-4	
DPLECB DS	0F	ECB PARAMETER
DPLIECB DS	0F	INTERNAL ECB
DPLXECB DS	0A	EXTERNAL ECB ADDRESS

DPLEXIT	DS	0A	EXIT ROUTINE ADDRESS
	ORG	*+4	
	EJECT		
DPLOPTCD	DS	F	OPTION CODES
	ORG	DPLOPTCD	
DPLOPCD1	DS	X	OPTION CODE #1
DOASYNC	EQU	B'10000000'	OPTCD=SYNC ASYNC
DONOCOPY	EQU	B'01000000'	OPTCD=COPY ORIGINAL
DOLOCAL	EQU	B'00100000'	OPTCD=GLOBAL LOCAL
DONOBLOK	EQU	B'00010000'	OPTCD=BLOCK NOBLOCK
*	EQU	B'00001000'	RESERVED
*	EQU	B'00000100'	RESERVED
*	EQU	B'00000010'	RESERVED
*	EQU	B'00000001'	RESERVED
DPLOPCD2	DS	X	OPTION CODE #2
DPLOPCD3	DS	X	OPTION CODE #3
DPLOPCD4	DS	X	OPTION CODE #4
DPLRTNCD	DS	F	COMPOSITE RETURN CODE
	ORG	DPLRTNCD	
DPLACTCD	DS	X	RECOVERY ACTION CODE
DAOKAY	EQU	0	SUCCESSFUL COMPLETION
DAEXCPTN	EQU	4	EXCEPTIONAL CONDITION
DAENVIRO	EQU	8	ENVIRONMENTAL CONDITION
DAFORMAT	EQU	12	FORMAT OR SPECIF ERRORS
DAPROCED	EQU	16	SEQUENCE AND PROCED ERRORS
DADPLERR	EQU	20	LOGIC ERRORS W/NO DPL RTNCD
DPLERRCD	DS	X	SPECIFIC ERROR CODE
DCOKAY	EQU	B'00000000'	00: NO CONDITIONALS
DCMORE	EQU	B'10000000'	00: MORE OCCUR. THAN SIZE
DCALIAS	EQU	B'01000000'	00: NAME WAS AN ALIAS
DCOVRFLO	EQU	B'00100000'	00: QNBUF TOO SMALL
DCNAMEIA	EQU	B'00010000'	00: NAME WAS INT ADDRESS
DCLOCAL	EQU	B'00001000'	00: GLOBAL USED LOCAL
DENONAME	EQU	1	04: NAME BUFFER/LEN ZERO
DENOVALU	EQU	2	04: VALUE BUFFER/LEN ZERO
DENOQNAM	EQU	3	04: QUAL NAME LEN ZERO
DETIMOUT	EQU	4	04: REQUEST TIMED OUT
DERFAIL	EQU	5	04: RESOLVER PROCESS FAILED
DENOTFND	EQU	6	04: REQUEST NOT FOUND
DENOCDS	EQU	7	04: NO CONFIG DATASET
DENAMERR	EQU	8	04: NAME DOES NOT EXIST
DEOVRFLO	EQU	9	04: RESULT BUFFER TOO SMALL
DENOBLOK	EQU	10	04: NOT READILY AVAILABLE
DENODATA	EQU	11	04: SERVER HAS NO DATA
DENAMODE	EQU	12	04: 31B NABUF/AMODE 24
DEVAMODE	EQU	13	04: 31B VABUF/AMODE 24
DEQNMODE	EQU	14	04: 31B QNBUF/AMODE 24
DESYSERR	EQU	1	08: SYSTEM ERROR
DESUBSYS	EQU	2	08: SUBSYSTEM ERROR
DENOTCNF	EQU	3	08: SUBSYS NOT INSTALLED
DENOTACT	EQU	4	08: SUBSYS NOT ACTIVE
DENOTRDY	EQU	5	08: SUBSYS NOT INITIALIZED
DESTOP	EQU	6	08: SUBSYS IS STOPPING
DEUNAVBL	EQU	7	08: UNAVAIL SERV/FACILITY
DERSOURC	EQU	8	08: INSUFFICIENT RESOURCES
DENOTPRB	EQU	9	08: USER RB NOT PRB
DETERM	EQU	10	08: SUBSYS HAS TERMINATED
DEBDOPCD	EQU	1	12: INVALID OPTION CODE
DEBDFNCD	EQU	2	12: INVALID FUNCTION CODE
DEBDXECB	EQU	3	12: INVALID ECB ADDRESS
DEBDEXIT	EQU	4	12: INVALID EXIT ADDRESS
DEBDNAME	EQU	5	12: INVALID DOMAIN NAME
DEBDVALU	EQU	6	12: INVALID VALUE
DEBDQNAM	EQU	7	12: INVALID QUAL NAME BUFF
DEACTIVE	EQU	1	16: DPL IS STILL ACTIVE
DEBDTYPE	EQU	1	20: DPL STNDRD FORM NOT 237

```

DEPROTCT EQU 2          20: DPL IS FETCH/STORE PROT
DEPLMODE EQU 3          20: 31B DPL PTR/AMODE 24
DPLDGNCD DS H           DIAGNOSTIC AND SENSE CODES
DPLFXPAR DS XL(12) 4X3  FIXED-LENGTH (INTERNAL) PARMS
                   ORG DPLFXPAR
DPLTIME DS F           TIME LIMIT
DPLSIZE DS F           SIZE LIMIT
                   DS F           RESERVED
DPLVAPAR DS XL(24) 8X3  VARIABLE-LENGTH (EXTERNAL) PARMS
                   ORG DPLVAPAR
DPLNAME DS XL(8) 4X2   PURPORTED NAME
                   ORG DPLNAME
DPLNABUF DS A           PARAMETER ADDRESS
DPLNALEN DS F           PARAMETER LENGTH
DPLVALUE DS XL(8) 4X2  VALUE PARAMETER
                   ORG DPLVALUE
DPLVABUF DS A           PARAMETER ADDRESS
DPLVALEN DS F           PARAMETER LENGTH
DPLQNAME DS XL(8) 4X2  FULLY QUALIFIED NAME
                   ORG DPLQNAME
DPLQNBUF DS A           PARAMETER ADDRESS
DPLQLEN DS F           PARAMETER LENGTH
DPLLEN EQU *-DPL       LENGTH OF DPL
* GENERAL RETURN CODES (RETURNED IN R15) ARE USED TO INDICATE
* SUCCESSFUL OR UNSUCCESSFUL COMPLETION OF A REQUEST IN
* SYNCHRONOUS MODE, AND ACCEPTANCE OR NON-ACCEPTANCE OF A
* REQUEST IN ASYNCHRONOUS MODE.
DROKAY EQU 0           SUCCESSFUL COMPLETION/ACCEPTED
DRFAILED EQU 4         UNSUCCESS COMPLETION/NOT ACCEPT DRFATLPL
                   EQU 8         FATAL DPL ERROR

```

C Language Definition

This section contains the C language definition for dpl, the Directory Services Parameter List.

```

#define DS_H
/*
 * exit address or event control block union
 */
union dplecbexit
{
    union
    {
        unsigned long    dpliecb;    /* internal ECB */
        unsigned char    *dplxecb;   /* external ECB address */
    } dplecb;
    void (*dplexit)();
/* address of exit routine */
};
/*
 * option codes
 */
union dploptcd
{
    unsigned long    dploptcdl;    /* options as long */
    struct
    {
        unsigned char    dplopcd1;    /* options as bytes */
        unsigned char    dplopcd2;    /* options as bytes */
        unsigned char    dplopcd3;    /* options as bytes */
        unsigned char    dplopcd4;    /* options as bytes */
    } dploptcds;
}

```

```

};
/*
 * return code field union
 */
union dplrtncd
{
    unsigned long    dplrtncdl;    /* return completion stat*/
    struct
    {
        unsigned char    dplactcd;    /*recovery action code */
        unsigned char    dplerrcd;    /*specific error code */
        unsigned short    dpldgncd;    /*diagnostic sense codes */
    }    dplrtncds;
};
/*
 * fixed-length parameters for dpl
 */
struct dplfxpar
{
    unsigned long        dpltime;        /* time limit */
    unsigned long        dplsize;        /* size limit */
    unsigned long        dplfxrsvd;    /* reserved */
};
/*
 * variable-length parameters for purported name
 */
struct dplname
{
    char                *dplnabuf;    /* parameter address */
    unsigned long        dplnalen;    /* parameter length */
};
/*
 * variable-length parameters for value parameters
 */
struct dplvalue
{
    char                *dplvabuf;    /* parameter address */
    unsigned long        dplvalen;    /* parameter length */
};
/*
 * variable-length parameters for fully qualified name
 */
struct dplqname
{
    char                *dplqnbu;    /* parameter address */
    unsigned long        dplqnl;    /* parameter length */
};
/*
 * variable-length parameters
 */
struct dplvapar
{
    struct dplname        dplname;    /* purported name */
    struct dplvalue        dplvalue;    /* value parameter*/
    struct dplqname        dplqname;    /* fully qualif. name */
};
/*
 * DNR request structure for calling the DIRSRV() function.
 */
struct dpl
{
    unsigned char        dplident;    /* control block ID */
    unsigned char        dplfnccd;    /* func code of req */
    unsigned char        dplactiv;    /* activity semaphore */
    unsigned char        dplflags;    /* flags used by API */
    unsigned char        dplsysid[4];    /* MVS subsystem ID */
};

```

```

        union dplecbexit dplecbexit;    /* ecb/exit rtn addr */
        union dploptcd  dploptcd;      /* option codes */
        union dplrtncd  dplrtncd;     /* return codes */
        struct dplfxpar  dplfxpar;     /* fixed length parms */
        struct dplvapar  dplvapar;     /* variable length parms */
};
/*
 * dplident
 */
#define DPLIDSTD 237
/*
 * dplfnccd
 */
#define DFHSTBYN 1 /* get host by name */
#define DFHSTBYV 2 /* get host by value */
#define DFHSTBYA 3 /* get host by alias */
#define DFNETBYN 4 /* get network by name */
#define DFNETBYV 5 /* get network by value*/
#define DFSRVBYN 6 /* get service by name */
#define DFSRVBYV 7 /* get service by value*/
#define DFPROBYN 8 /* get protocol by name*/
#define DFPROBYV 9 /* get protocol by value */
#define DFHSTSRV 10 /* get services by name*/
#define DFHSTINF 11 /* get host info by name */
#define DFRROUTE 12 /* get mail route */
#define DFRPCBYN 13 /* get rpc by name */
#define DFRPCBYV 14 /* get rpc by value */
#define DFMAX DFRPCBYV /* MAXIMUM function code */
/*
 * dplflags
 */
#define DPLFCMPL 0x80 /* DPL completed */
#define DPLFCERR 0x40 /* Completed with error*/
#define DPLFXECB 0x20 /* external ECB in use */
#define DPLFEXIT 0x10 /* exit in use */
#define DPLF31B 0x08 /* req iss in AMODE=31 */
/*
 * dplpcd1
 */
#define DOASYNC 0x80 /* OPTCD=SYNC | ASYNC */
#define DONOCOPY 0x40 /* OPTCD=COPY | ORIGINAL */
#define DOLOCAL 0x20 /* OPTCD=GLOBAL | LOCAL*/
#define DONOBLOK 0x10 /* OPTCD=BLOCK | NOBLOCK*/
/*
 * dplrtncd.dplactcd
 */
#define DAOKAY 0 /* successful completion */
#define DAEXCPTN 4 /* exceptional condition */
#define DAENVIRO 8 /* environmental cond */
#define DAFORMAT 12 /* format/specif error */
#define DAPROCED 16 /* sequence/proced error */
#define DADPLERR 20 /* logic error (no rtn code)*/
/*
 * dplrtncd.dplerrcd
 */
#define DCOKAY 0x00 /* 00: no conditionals */
#define DCMORE 0x80 /* 00: more occr thn siz */
#define DCALIAS 0x40 /* 00: name was alias */
#define DCOVRFLO 0x20 /* 00: QNBUF too small */
#define DCNAMEIA 0x10 /* 00: name was IA */
#define DCLOCAL 0x08 /* 00: global used local */
#define DENONAME 1 /* 04: name buffer or len = 0 */
#define DENOVALU 2 /* 04: val buffer or len = 0 */
#define DENOQNAM 3 /* 04: qual name len = 0 */
#define DETIMOUT 4 /* 04: request timed out */
#define DERFAIL 5 /* 04: resolver failure*/

```

```

#define DENOTFND      6      /* 04: request not found */
#define DENOCDS      7      /* 04: no config data set */
#define DENAMERR     8      /* 04: name doesn't exist */
#define DEOVRFLO     9      /* 04: result buff too small */
#define DENOBLOK    10     /* 04: not readily avail */
#define DENODATA    11     /* 04: server has no data */
#define DENAMODE    12     /* 04: 31B nabuf/amode 24 */
#define DEVAMODE    13     /* 04: 31B vabuf/amode 24 */
#define DEQNMODE    14     /* 04: 31B qnbuf/amode 24 */
#define DESYSERR     1      /* 08: system error */
#define DESUBSYS     2      /* 08: subsystem error */
#define DENOTCNF     3      /* 08: susbsys not config */
#define DENOTACT     4      /* 08: susbsys not active */
#define DENOTRDY     5      /* 08: susbsys not initialized*/
#define DESTOP       6      /* 08: susbsys is stopping */
#define DEUNAVBL     7      /* 08: unavai service/facil */
#define DERSOURCE    8      /* 08: insufficient resources */
#define DENOTPRB     9      /* 08: user RB not PRB */
#define DETERM      10     /* 08: subsystem has stopped */
#define DEBDOPCD     1      /* 12: invalid option code */
#define DEBDFNCD     2      /* 12: invalid function cd */
#define DEBDXECB     3      /* 12: invalid ecb address */
#define DEBDEXIT     4      /* 12: invalid exit address */
#define DEBDNAME     5      /* 12: invalid domain name */
#define DEBDVALU     6      /* 12: invalid value */
#define DEBDQNAM     7      /* 12: inval qualified name */
#define DEACTIVE     1      /* 16: DPL is active */
#define DEBDTPYE     1      /* 20: inval DPL identifier */
#define DEPROTCT     2      /* 20: fetch/store protect */
#define DEPLMOD      3      /* 20: 31B DPL/amode 24 */
/*
 * general return codes
 */
#define DROKAY       0      /* successful completion */
#define DRFAILED     4      /* unsuccess completion */
#define DRFATLPL     8      /* fatal DPL error */

```


Configuration

This chapter contains information on installation, configuration, and customization of the API and C socket libraries. It includes the following sections:

- [Socket and C Library Installation](#) – Describes the installation of the socket and C libraries
- [Socket Configuration](#) – Describes the configuration file and sockcfg configuration structure. Explains socket life and buffering limits. Also describes compiling the configuration file
- [Sample JCL for Compiling Socket Programs](#) – Contains sample JCL for compiling and linking the socket configuration file. Includes the IBM and SAS/C sockcfg configuration changes
- [Using Socket Libraries](#) – Includes sample JCL showing how to compile, link, and execute non-reentrant and reentrant C socket programs using IBM C/370 and SAS/C compilers
- [Customizing Socket Programs](#) – Describes how to change default values within the socket program

Socket and C Library Installation

This section describes the installation and configuration of the API, socket, and C libraries. For additional explanation of the socket and basic C library functions and features refer to the chapters “C Library Functions” and the “Socket Library Functions.”

The installation of the socket and basic C libraries is accomplished during the SMP installation of the API. The socket and C library object, load, and header files are created by the SMP APPLY process.

Library Data Sets

The socket and C libraries are provided in a number of Partitioned Data Set (PDS) libraries. These libraries are explained in “[Header File Library](#).”

The following table outlines the socket and C library data sets created during SMP APPLY processing

Data Set Name	SMP Name	Description
H	H	C include (.h) files (members).
CILIB	APICIL	Subroutine library in load module form for non-reentrant IBM C/370 users.
CSLIB	APICSL	Subroutine library in load module form for non-reentrant SAS/C users.
CIROBJ	APICIRO	Subroutine library in object form for reentrant IBM C/370 users.
CSROBJ	APICSRO	Subroutine library in object form for reentrant SAS/C users.
SAMP	TCPSAMP	sockcf.c source as SOCKCFG member.
CNTL	CNTL	Sample JCL to compile SOCKCFG and to compile, link, and execute a socket program.

Header File Library

The header file library contains the header files you need to include to use the socket library. You must include the correct header files in your C source code before compiling to ensure proper compilation and execution of the socket library. Header files that are no longer necessary are noted but included for backwards compatibility. The following table lists and describes the header files:

Header File	Definition
acs.h	This header file is no longer required, but is included for backwards compatibility.
api.h	This header file is for use with the C library and when building the socket configuration file. It is not needed to interface to the socket library. It describes the interface to the API when using the C library.
cdefs.h	This header file contains common definitions for compilation of C headers.
ds.h	This header file is for use with the C library. It is not needed to interface to the socket library. It describes the interface to the Network Directory Services (NDS) when using the C library.
ecb.h	This header file is needed when compiling the socket configuration file and is also used internally by the socket library. It is not needed to interface to the socket library.
errno.h	This header file defines the errors that can be returned by the socket library when making requests to it. It also defines GET_ERRNO for backwards compatibility.
ibmc.h	This header file is needed when compiling the socket configuration file and is also used internally by the socket library. It is not needed to interface to the socket library.
icssckt.h	This header file is used for Unicenter TCPAccess UNIX System Services (formerly OpenEdition) sockets. See the chapter "UNIX System Services MVS Integrated Sockets" for more information.
if.h	This header file defines the interface control blocks and ioctl request formats.
if_arp.h	This header file defines the ARP header.
if_ether.h	This header file defines ethernet frame formats.
in.h	This header file defines the constants and structures necessary to interface to a socket in the internet domain.
inet.h	This header file includes in.h and several macro definitions.
iocomm.h	This header file contains ioctl encoding definitions.
ioctl.h	This header file contains standard ioctl definitions for both files and sockets.

Header File	Definition
ip.h	This header file defines values used by the Internet Protocol (IP) and details the format of an IP header and options associated with IP. The current implementation of the socket library does not let the socket library user access the IP layer; therefore this header file is of little use at this time. This header file will be used in later versions of the socket library that will let the socket library user set IP options.
netdb.h	This header file defines the structures used by the <i>get</i> services. It also provides the function prototypes for this family of functions.
param.h	This header file contains various system parameters and macros.
proto.h	This header file is needed when compiling the socket configuration file and is also used internally by the socket library. It is not needed to interface to the socket library.
sasc.h	This header file is needed when compiling the socket configuration file and is also used internally by the socket library. It is not needed to interface to the socket library.
serrno.h	This header file is no longer required and simply includes <i>errno.h</i> and <i>sockcfg.h</i> . It is provided for backwards compatibility.
sockcfg.h	This header file describes the socket configuration structure. It is covered in more detail later in this chapter.
socket.h	This header file defines most of the variables and structures required to interface properly to the socket library. It also provides the function prototypes in ANSI form for those functions that are truly socket functions.
sockio.h	This header file defines socket <i>ioctl</i> values. It includes <i>ioctl.h</i> .
sockvar.h	This header file is needed when compiling the socket configuration file and is also used internally by the socket library. It is not needed to interface to the socket library.
syslog.h	This header file contains header and prototype information for the emulation of a <i>syslog</i> daemon.
system.h	This header file contains system time structs and defines.
tcp.h	This header file describes those options that can be set for a socket of type <i>SOCK_STREAM</i> .
time.h	This header file contains time structs and defines. A special note is included for SAS/C users concerning the use of the define <i>_USE_SASC_TIME</i> .
types.h	This header file contains various typedefs required for using the socket library. This header file contains most of the BSD <i>types.h</i> with exceptions for items inconsistent with the MVS environment.

Header File	Definition
uio.h	This header file describes the structures necessary to use vectored buffering of the socket library. Due to ANSI checking, this header file must be included in any header file that includes socket.h.
unistd.h	This header file contains standard UNIX defines minus the standard UNIX function prototypes.
user.h	This header file is needed when compiling the socket configuration file and is also used internally by the socket library. It is not needed to interface to the socket library.

Load Library

The socket and C library routines are provided as a library of load modules. The load library is for use in construction of non-reentrant applications; reentrant program developers should use the object library discussed in the next section. Each compiler has its own load library. When users link their code, they should choose the proper library based on the compiler. The load library should be specified in the SYSLIB concatenation for the link-edit step.

Object Library

To support the creation of reentrant load modules, the socket and C library routines are also supplied as object modules. Each compiler (IBM and SAS) has one object library. When users pre-link their reentrant code, they should choose the proper library based on the compiler used to compile the program. The object library should be specified in the SYSLIB concatenation for the compiler pre-link step.

Sample JCL

Sample JCL for using the socket library is provided in the CNTL data set.

The related members are:

CLGIBMC	Sample JCL to compile, link, and execute a non-reentrant user program using the IBM C/370 C compiler.
CLGIBMCR	Sample JCL to compile, link, and execute a reentrant user program using the IBM C/370 C compiler.
CLGSASC	Sample JCL to compile, link, and execute a non-reentrant user program using the SAS/C compiler.
CLGSASCR	Sample JCL to compile, link, and execute a reentrant user program using the SAS/C compiler.
UMODCFGI	Sample JCL to compile and link customized sockcf.c for IBM C/370 C users.
UMODCFGS	Sample JCL to compile and link customized sockcf.c for SAS/C users.

Note: You should run an SMP/E APPLY CHECK against any USERMOD that you are trying to install as there may be additional PREs on your system that are not accounted for. Once you gather this information, add the SYSMOD list(s) to the ++PRE(#####) statement. Then SMP/E REJECT the USERMOD to remove the invalid entry from the SMP/E CSI. You will then be able to RECEIVE/APPLY the USERMOD with success.

Socket Configuration

Once the libraries are created by SMP, the socket library can be configured to meet the needs and dependencies of the site and its users; the C library routines need no customization. The socket library can be customized by each user or by the site system administrator to enforce resource constraints on the users of the socket library. The configuration is accomplished by modifying a configuration file and then compiling it into object module and load module form. The object module is then placed in the data set that has been designated for use as the socket library reentrant object data set. The load module is placed in the data set designated for use as the socket library non-reentrant data set. If you want a custom copy of the configuration, follow the same steps but copy the object and load modules to your private libraries. The source for sockcf.c is in the SAMP data set and the header file (sockcfg.h) is in the H data set.

Configuration File

The configuration file is provided in C source code compilable by either the SAS/C or IBM C compilers. The file's name is sockcf.c and its partner header file, sockcfg.h, describes the structure and default settings of the configuration.

sockcfg Configuration Structure

The configuration is accomplished by changing any configuration options in the sockcf file and then recompiling it. Each variable of the configuration structure is listed below with the allowable range within which it may be set. To change parameters without recompiling the socket file, see t6:

Configuration Options	Definition
sockcfg.name [4]	An array containing the characters CNFG. This is used as an eye-catcher and should never be changed during socket library configuration.
sockcfg.length	This integer contains the length in bytes of the configuration structure. Like the previous variable, this one should never be changed by the user during configuration.
sockcfg.nosockets	Specifies the maximum number of sockets that each user can use at any one time. This variable can be used to restrict or limit the usage of resources of the socket user. The minimum is 1 and the maximum is 512. The limits placed on this variable are checked (if CONFIGDEBUG is set in sockcfg.flags) at socket library startup time. Other limits may be placed on this variable by setting the sockcfg.maxsocket and sockcfg.minsocket variables. See " Socket Configuration " for more information. Default: 20.

Configuration Options	Definition
sockcfg.maxsocket	Maximum number of sockets to let a user specify in the socket configuration. This is only a soft limit and may be varied within the hard limits previously discussed. Its primary use is to check the parameters configured by the user. Default: 512.
sockcfg.minsocket	Minimum number of sockets a user may specify in the socket configuration. Its primary use is to check the parameters configured by the user. Default: 20.
sockcfg.fdssocket	Sets the base number that the socket library can use to assign file descriptors to sockets as they are opened. This value must be greater than the maximum file descriptor used by the native C compiler library for files. In this way, the socket library can distinguish between a socket or a file when a function that is supported by both the native C library and the socket library is called. Such functions as read, write, close, and others can be performed by both files and sockets. The current SAS/C compiler uses file descriptors below 40 and the current IBM C compiler does not support operations on file descriptors. The minimum is zero and the maximum can be calculated by taking the minimum of 512 and sockcfg.maxsocket and subtracting sockcfg.nosockets. For almost all uses the default should suffice. See " Socket Configuration " for more information. Default: 64.
sockcfg.maxsndbytcnt	Sets the maximum transmit byte count that a user may request when setting the send byte buffering characteristics of a socket. This limit cannot exceed the API site default for send byte buffering. The socket library uses the smaller of the two values as the maximum send byte count. Default: 64000.
sockcfg.minsndbytcnt	Sets the minimum transmit byte count that users may request when setting the send byte buffering characteristics of a socket. The smallest value allowed is 128. Default: 128.
sockcfg.defsndbytcnt	Defines the default send byte buffer allocation of each socket. This value must be between the maximum and minimum values. If USEAPIREQBYTCNTDEFS is set in sockcfg.flags, this variable is ignored and the default value assigned by the API is used. Default: 32000.
sockcfg.maxsndreqcnt	Sets the maximum transmit request count that users may request when setting the send request buffering characteristic of a socket. This limit cannot exceed the API site default for send request buffering. The socket library uses the smaller of the two values as the maximum. Default: Eight.

Configuration Options	Definition
sockcfg.minsndreqcnt	Sets the minimum transmit request count that users may request when setting the send request buffering characteristics of a socket. Default: One.
sockcfg.defsndreqcnt	Defines the default send request buffer allocation of each socket. This value must be between the maximum and minimum values. If USEAPIREQBYTCNTDEFS is set in sockcfg.flags, this variable is ignored and the default value assigned by the API is used. Default: Four.
sockcfg.maxrcvbytcnt	Sets the maximum receive byte count that users may request when setting the receive byte buffering characteristics of a socket. This limit cannot exceed the API site default for receive byte buffering. The socket library uses the smaller of the two values as the maximum. Default: 64000.
sockcfg.minrcvbytcnt	Minimum receive byte count that users may request when setting the receive byte buffering characteristics of a socket. The smallest value allowed is 128. Default: 128.
sockcfg.defrcvbytcnt	Defines the default receive byte buffer allocation of each socket. This value must be between the maximum and minimum values. If USEAPIREQBYTCNTDEFS is set in sockcfg.flags, this variable is ignored and the default value assigned by the API is used. Default: 32000.
sockcfg.maxrcvreqcnt	Sets the maximum receive request count that users may request when setting the receive request buffering characteristics of a socket. This limit cannot exceed the API site default for receive request buffering. The socket library uses the smaller of the two values as the maximum. Default: Eight.
sockcfg.minrcvreqcnt	Sets the minimum receive request count that users may request when setting the receive request buffering characteristics of a socket. Default: One.
sockcfg.defrcvreqcnt	Defines the default receive request buffer allocation of each socket. This value must be between the maximum and minimum values. If USEAPIREQBYTCNTDEFS is set in sockcfg.flags, this variable is ignored and the default value assigned by the API is used. Default: Four.

Configuration Options	Definition
sockcfg.subsysid	This character string pointer should point to the four-character string that is the Unicenter TCPaccess subsystem identifier. This variable is the most common variable that each socket configuration must change. When configuration is to take place, the installer should determine what the subsystem ID is for Unicenter TCPaccess and then ensure this variable points to this string prior to compiling this module. An improperly specified subsystem ID is detected when the socket library attempts to open a session with the API on behalf of the socket user. This occurs on the first call to the socket function. See " Customizing Socket Programs " for more information. Default: ACSS.
sockcfg.svcid[2]	This array of pointers should point to the character strings that designate the services provided by the API. The defaults should suffice and, in most instances, the installer need not change these. Default: for TCP is TCP; for UDP is UDP.
sockcfg.apcbapplid	This variable should point to an application ID. Currently this is not supported and should therefore be set to point to a null string.
sockcfg.apcbpasswd	This variable should point to an application password. Currently, this is not supported and should therefore be set to point to a null string.
sockcfg.apcbflags	This character is used "as is" in the APCB on the application open (AOPEN) call to the assembler API. This value is placed in the apcbflag element of the APCB. For more details on the use of the apcbflag byte, see apcb in the chapter "." Default: Zero.
sockcfg.apcboptc	Used "as is" in the APCB on the application open call to the assembler API. This value is placed in the apcboptc element of the APCB. For more details on the use of the apcbflag byte, see apcb in the chapter "." Default: Zero.
sockcfg.errnobase	This integer sets the base value for the socket library to set errno errors. To avoid confusion with the native C library errors, this element should have a value greater than the maximum error value that the native C library would set in errno. The current SAS/C compiler runtime library uses error codes below 50. The current IBM C compiler library does so also. Default: 100
sockcfg.exitfunc	This variable is a pointer to the C library exit function that is used to set up the shutdown procedure. This should never need to be changed. Default: text().

Configuration Options	Definition
sockcfg.flags	<p>This unsigned long variable is a mask of flag bits that control the characteristics of the socket library. These bits are defined:</p> <p>31 - SIGNALSUPP</p> <p>If set to one, user-added signals for SIGIO, SIGURG, and SIGPIPE may be used if they are also defined. This bit is only valid with the SAS/C library.</p> <p>30 - USEAPIREQBYTECNTDEF</p> <p>If set to one, the socket library ignores the configuration of the data buffering characteristics and uses the API defaults. The default and recommended setting is to use the API defaults for buffering limits.</p> <p>29 - EXTERRNOMSGS</p> <p>If set to one, the perror function of the socket library prints extended error messages when it detects a socket configuration problem or a session initialization problem with the assembler API. When used in conjunction with the CONFIGDEBUG option, this option lets the installer configure the socket library and then run a test program to see if any errors are detected with the configuration. Once the configuration is completed successfully, this flag bit and CONFIGDEBUG bit can be turned off.</p> <p>0 - CONFIGDEBUG</p> <p>If set to one, user-added signals for SIGIO, SIGURG, and SIGPIPE may be used if they are also defined. This bit is only valid with the SAS/C library. Default: 0x80000007 for the SAS/C compiler, 0x80000006 for the IBM C compiler.</p>
sockcfg.closetimeout	<p>Sets the time in seconds that the socket library is to wait for an orderly release on a SOCK_STREAM socket to occur when the user issues a close library call in blocking mode. This timeout value is not used for closes issued in nonblocking mode so as not to affect the user's environment (rob the user of the one and only timer available). The closetimeout value can be overridden by using the SO_LINGER option of sockets.</p> <p>Default: 120 seconds (two minutes).</p>
sockcfg.envrinit	<p>This is a pointer to a function that does C library specific initialization. This should never have to be changed by the installer as the proper routines are included if the correct #defines are turned on at compile time. The routine for the SAS/C library is s0scinit() and for the IBM C library is s0icinit().</p>
sockcfg.envrterm	<p>This is a pointer to a function that does C library specific termination. This should never have to be changed by the installer as the proper routines are included if the correct #defines are turned on at compile time. The routine for the SAS/C library is s0scterm() and for the IBM C library is s0icterm().</p>

Configuration Options	Definition
sockcfg.comfuncs	This structure of function pointers can be used by the installer to specify the entry point to functions that are common to both the socket library and the native C runtime library. For the SAS/C compiler, the only functions that have duplicate names are read(), write(), and close(). The IBM C compiler has no functions that have duplicate names with functions of the socket library.
sockcfg.comfuncs.read	File descriptor or socket read function.
sockcfg.comfuncs.readv	File descriptor or socket read function using vectored I/O buffering.
sockcfg.comfuncs.write	File descriptor or socket write function.
sockcfg.comfuncs.writev	File descriptor or socket write function using vectored I/O buffering.
sockcfg.comfuncs.ioctl	I/O control function.
sockcfg.comfuncs.fcntl	File descriptor or socket control function.
sockcfg.comfuncs.select	File descriptor or socket I/O synchronous I/O multiplexing function.
sockcfg.comfuncs.close	File descriptor or socket close function.
sockcfg.sigurg	Signal number to use for signaling the reception of urgent (OOB) data on a socket. This is only supported with the SAS/C library and the signal used must be one of the user-assigned asynchronous signals. Default: For the SAS/C compiler is SIGASY6, for the IBM C compiler (for which this option is not supported) is zero.
sockcfg.sigio	Signal number to use for signaling the occurrence of a major event on a socket. Major events include the completion of a connection request done asynchronously, an incoming connection request on a server socket, the abnormal termination of a connection, the reception of an orderly release from the remote endpoint, and the reception of regular data for a socket or any asynchronous error that may occur during the life of a socket. This is only supported with the SAS/C library and the signal used must be one of the user-assigned asynchronous signals. Default: For the SAS/C compiler is SIGASY7, for the IBM C compiler (for which this option is not supported) is zero.
sockcfg.sigpipe	Signal number to use for signaling an error when a socket that can no longer send any more data has a write request issued to it. This is only supported with the SAS/C library and the signal used must be one of the user-assigned asynchronous signals. Default: For the SAS/C compiler is SIGASY8, for the IBM C compiler (for which this option is not supported) is zero.

Socket Life

A socket is created by the successful completion of the socket library call. It remains in existence until closed explicitly via the close library call or implicitly via task or address space termination by the socket library user. For sockets of the SOCK_STREAM type that use TCP for transport, close induces TCP to perform an orderly release. This process involves the local endpoint sending a FIN to the remote endpoint and then receiving an ACK for its FIN and a FIN from the remote endpoint. This process may take a great amount of time (as compared to computer instruction execution time).

If the user is running in nonblocking I/O, the close call returns once the local endpoint has scheduled the transmission of the FIN. The socket can no longer be acted on by the socket user but the context related to the socket is still counted against the user's quota of sockets. Not until the orderly release completes does this socket get deallocated and allow for its reuse. If a user does not desire such a feature, the SO_LINGER option may be set so that the socket is freed up possibly prior to the completion of the orderly release. If this option is to be used, the socket user must be prepared for the consequences related to the lack of a proper release occurring at the remote endpoint.

Socket Buffering Limits

The configuration variables listed in [Compiling the Configuration File](#) define the limits on each socket's receive and transmit buffering abilities. Sockets do not own any data buffers; the underlying assembler API controls these resources. However, sockets can be used to inform the assembler API how much buffering is desired, thus letting the assembler API allocate the appropriate amount of resources to satisfy users' requirements.

Instead of using the API defaults, socket users can have greater restrictions placed on them by configuring the parameters listed in [Socket Configuration](#) accordingly. In all cases, socket users are not able to acquire more than the API maximums. If the defaults provided in the configuration are not enough to satisfy the socket users, they may request more by using the setsockopt library call.

Receive and transmit buffering are totally independent of one another. Buffering limits consist of these elements:

- Number of bytes for actual data to allocate
- Number of send or receive requests that the socket library can have internally pending at a given time

This is much like UNIX sockets, which provide a certain number of bytes for buffering while at the same time limiting the number of mbufs a user can tie up at a given time.

We recommend that the USEAPIREQBYTECNTDEFS be set on in sockcfg.flags; this provides socket users with the default site parameters for API buffering characteristics. The only time this flag should be turned off is when resource restrictions are to be placed on socket library users.

Compiling the Configuration File

Once the configuration file has been modified for site dependencies, it must be properly compiled. Sample JCL is provided in the CNTL data set to accomplish this task. The name of the member is UMODCFGI for IBM C/370 users and UMODCFGS for SAS/C users. This JCL creates the sockcf module for both reentrant and non-reentrant usage. The most important part of this task is to ensure the proper PARMs are used at compilation time. The following table lists the PARMs of importance:

PARM	Comments
DEF (SASC)	Must be used only with the SAS/C compiler.
DEF (IBMC)	Must be used only with the IBM C compiler.
RENT	Must be specified when generating the reentrant version of the code and should not be used when generating the non-reentrant version of the code.

The PARM field in each sample JCL is shipped with the proper configuration. This table outlines the necessary PARMs required to build each socket configuration module type:

Compiler	Code Type	DEF(SASC)	DEF(IBM C)	RENT
SAS/C	non-reentrant	X		
SAS/C	reentrant	X		X
IBM C	non-reentrant		X	
IBM C	reentrant		X	X

Note: When building the load module form of the configuration file, these names are unresolved at single module link time:

For SAS/C: READ, WRITE, CLOSE, ATEXT, S0SCINT, S0SCTRM
 For IBM C: S0ICINT, S0ICTRM

These variables are resolved when the user program is linked with the socket library.

You should run an SMP/E APPLY CHECK against any USERMOD that you are trying to install as there may be additional PREs on your system that are not accounted for. Once you gather this information, add the SYSMOD list(s) to the ++PRE(#####) statement. Then SMP/E REJECT the USERMOD to remove the invalid entry from the SMP/E CSI. You will then be able to RECEIVE/APPLY the USERMOD with success.

Sample JCL for Compiling Socket Programs

This section contains sample JCL for compiling and linking the socket configuration file. It includes examples for users of IBM C/370 and SAS/C compilers.

Note: If you are link-editing with the BINDER (HEWLF096) under SMP/E, you may get the error message IEW2480W. This message can be safely ignored. You can turn this message off by setting option MSGLEVEL=4 in the PARM field of the linkedit (binder).

IBM sockcfg Configuration Changes

```
//UMODCFG JOB
//*
//* SAMPLE SMP/E JCL TO RECEIVE AND APPLY A USERMOD TO
//* INSTALL A CUSTOMIZED VERSION OF SOCKCF SOURCE
//* CONTAINING C SOCKET CONFIGURATION PARAMETERS FOR
//* IBM C/370 USERS.
//*
//* EDIT THE JOB JCL STATEMENT AND VERIFY THAT THE DATA SET
//* NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
//* FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED
//* ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB
//* ASSUMES THAT THE STANDARD IBM 370 EDCC JCL PROCEDURES IS
//* AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
//*
//* GLOBALLY CHANGE THE FOLLOWING STRINGS TO CORRECTLY
//* REFLECT TARGET DATASETS AND SMPE FMID.
//*
//* 'TRGINDX' < SMPE TARGET DATASETS HIGH LEVEL QUALIFIER
//* XXX      < SMPE FMID IDENTIFYING MVS TCP/IP FMID
//*
//* STEP 1: COMPILE THE NON-REENTRANT VERSION OF SOCKCF
//* TO CREATE OBJECT WHICH WILL BE PASSED TO THE
//* SUBSEQUENT SMP/E USERMOD RECEIVE/APPLY STEP.
//*
//CNORENT      EXEC EDCC,CPARM='DEF(IBM),SEQUENCE(73,*)',
//              INFILE='TRGINDX.SAMP(SOCKCF)' <=== VERIFY DSNNAME
//COMPILE.SYSLIB DD
//              DD DISP=SHR,DSN=TRGINDX.H <=== VERIFY DSNNAME
//COMPILE.SYSLIN DD DISP=(NEW,PASS),DSN=&&COBJ(SOCKCFI),
//              UNIT=VIO,SPACE=(3200,(10,10,1)),
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//*
//* STEP 2: COMPILE THE REENTRANT VERSION OF SOCKCF
//* TO CREATE OBJECT WHICH WILL BE PASSED TO THE
//* SUBSEQUENT SMP/E USERMOD RECEIVE/APPLY STEP.
```

```

/*
//CRENT EXEC EDCC,CPARM='DEF(IBM),RENT,SEQUENCE(73,*)',
// INFIL='TRGINDX.SAMP(SOCKCF)' <=== VERIFY DSNAME
//COMPILE.SYSLIB DD
// DD DISP=SHR,DSN=TRGINDX.H <=== VERIFY DSNAME
//COMPILE.SYSLIN DD DISP=(OLD,PASS),DSN=&&COBJ(SOCKCFIR)
/*
/* STEP 3: EXECUTE SMP/E TO INSTALL A USERMOD TO UPDATE
/* THE REENTRANT AND NON-REENTRANT VERSIONS OF
/* SOCKCF FOR IBM C/370 USERS.
/*
/* CHANGE 'XXX' TO THE CORRECT MVS TCP/IP FMID.
/*
//SMPE EXEC PGM=GIMSMP,REGION=4096K,TIME=960,
// PARM='CSI=SMPINDX.CSI,PROCESS=WAIT'
//SMPHOLD DD DUMMY
//SMPLOG DD DSN=SMPINDX.SMPLOG,DISP=MOD
//SMPOUT DD SYSOUT=HOLDCL
//COBJ DD DISP=(OLD,DELETE),DSN=&&COBJ
//SMPPTFIN DD *
++ USERMOD (MU0CFGI) .
++ VER (Z038)
FMID(API0XXX) /* CHANGE TO YOUR MVS TCP/IP FMID */ .
++ USER2 (SOCKCFIR) TXLIB(COBI) DISTLIB(AAPICIRO) .
++ MOD (SOCKCFI) TXLIB(COBI) DISTLIB(AAPICL) .
/*
//SMPCNTL DD *
SET BDY(GLOBAL) .
RECEIVE S(MU0CFGI) .
SET BDY(TCPTZN) .
APPLY S(MU0CFGI) .
/*

```

SAS/C sockcfg Configuration Changes

```

//UMODCFGS JOB
/**
/** SAMPLE SMP/E JCL TO RECEIVE AND APPLY A USERMOD TO
/** INSTALL A CUSTOMIZED VERSION OF SOCKCF SOURCE
/** CONTAINING C SOCKET CONFIGURATION PARAMETERS FOR
/** SAS SAS/C USERS.
/**
/** EDIT THE JOB JCL STATEMENT AND VERIFY THAT THE DATA SET
/** NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
/** FOR THE MVS TCP/IP TARGET DATA SETS (DSN'S TO BE
/** VERIFIED ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS
/** JOB ASSUMES THAT THE STANDARD SAS/C LC370C JCL PROCEDURE
/** IS AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
/**
/** GLOBALLY CHANGE THE FOLLOWING STRINGS TO CORRECTLY
/** REFLECT TARGET DATASETS AND SMPE FMID.
/**
/** 'TRGINDX' < SMPE TARGET DATASETS HIGH LEVEL QUALIFIER
/** XXX      < SMPE FMID IDENTIFYING MVS TCP/IP FMID
/**
/** STEP 1: COMPILE THE NON-REENTRANT VERSION OF SOCKCF
/** TO CREATE OBJECT WHICH WILL BE PASSED TO THE
/** SUBSEQUENT SMP/E USERMOD RECEIVE/APPLY STEP.
/**
//CNORENT EXEC LC370C,PARM.C='DEF(SASC)'
//C.SYSLIN DD DISP=(NEW,PASS),DSN=&&COBJ(SOCKCFS),
// UNIT=VIO,SPACE=(3200,(10,10,1))
//C.SYSLIB DD DISP=SHR,DSN=TRGINDX.H          <=== VERIFY DSNAME
//          DD DISP=SHR,DSN=SASC.MACLIBC      <=== VERIFY DSNAME
//C.SYSIN  DD DISP=SHR,
//          DSN=TRGINDX.SAMP(SOCKCF)         <=== VERIFY DSNAME
/**
/** STEP 2: COMPILE THE REENTRANT VERSION OF SOCKCF TO
/** CREATE OBJECT WHICH WILL BE PASSED TO THE
/** SUBSEQUENT SMP/E USERMOD RECEIVE/APPLY STEP.
/**
//CRENT EXEC LC370C,PARM.C='DEF(SASC),RENT'
//C.SYSLIN DD DISP=(OLD,PASS),DSN=&&COBJ(SOCKCFSR)
//C.SYSLIB DD DISP=SHR,DSN=TRGINDX.H          <=== VERIFY DSNAME
//          DD DISP=SHR,DSN=SASC.MACLIBC      <=== VERIFY DSNAME
//C.SYSIN  DD DISP=SHR,
//          DSN=TRGINDX.SAMP(SOCKCF)         <=== VERIFY DSNAME
/**
/** STEP 3: EXECUTE SMP/E TO INSTALL A USERMOD TO UPDATE
/** THE REENTRANT AND NON-REENTRANT VERSIONS OF
/** SOCKCFG FOR SAS/C USERS.
/**
/** CHANGE THE 'XXX' TO THE CORRECT MVS TCP/IP FMID.
/**
//SMPE EXEC PGM=GIMSMP,REGION=4096K,TIME=960,
// PARM='CSI=SMPINDX.CSI,PROCESS=WAIT'
//SMPHOLD DD DUMMY
//SMPLOG DD DSN=SMPINDX.SMPLOG,DISP=MOD
//SMPDOUT DD SYSOUT=HOLDCL
//COBJ DD DISP=(OLD,DELETE),DSN=&&COBJ
//SMPPTFIN DD *
++ USERMOD (MU0CFGS) .
++ VER (Z038)
FMID(API0XXX) /* CHANGE TO YOUR MVS TCP/IP FMID */ .
++ USER1 (SOCKCFSR) TXLIB(COBJ) DISTLIB(AAPICSR) .
++ MOD (SOCKCFS) TXLIB(COBJ) DISTLIB(AAPICL) .
/**

```

```
//SMPCTL DD *  
SET BDY(GLOBAL) .  
RECEIVE S(MU0CFG5) .  
SET BDY(TCPTZN) .  
APPLY S(MU0CFG5) .  
/*
```

Using Socket Libraries

After the socket library has been properly installed, it is ready for use. When writing code to use the socket library, make sure the proper header files are included in each source file using socket structures, defines, or function calls. Once the code is written, concatenate the socket library header data set to their SYSLIB concatenation for the compile step.

For non-reentrant users, concatenate the proper socket library load data set to the SYSLIB concatenation for the link-edit step. For reentrant users, concatenate the proper socket library object data set to the SYSLIB concatenation and include the proper modules.

The sample JCLs show how to compile, link, and execute non-reentrant and reentrant C socket programs using the IBM C/370 and SAS/C compilers.

Compile/Link IBM C/370 C Non-reentrant Program

This JCL is for the C/370 version of the compiler. If you are using the AD/Cycle compiler, replace this line (shown in bold in the JCL):

```
// DD DISP=SHR,DSN=&VSCCHD&CVER&EDCHDRS
```

with this line:

```
// DD DISP=SHR,DSN=&LNGPRFX..SEDCHDR
```

The AD/Cycle compiler does not use the first DD statement shown after the LKED.SYSLIB DD in this JCL example. See sidebar comment.

```
//CLGIBMC JOB  
/*  
/* SAMPLE JCL TO COMPILE, LINK, AND EXECUTE A NONREENTRANT  
/* USER PROGRAM USING THE TCP/API C SOCKET LIBRARIES  
/* AND THE IBM C/370 C COMPILER.  
/*  
/* EDIT THE JOB JCL STATEMENT, VERIFY THE DATA SET NAME(S)  
/* OF THE USER'S DATA SETS, AND VERIFY THAT THE DATA SET  
/* NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED  
/* FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED  
/* ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB  
/* ASSUMES THAT THE STANDARD IBM C/370 EDCLG JCL PROCEDURE  
/* IS AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
```

```

/**
//CLGNRENT EXEC EDCCPLG,
//      INFILE='USER.C(CPROG)',           <=== VERIFY DSNAME
//      CPARAM='DEF(IBM),NORENT',
//      GPARM='PROGRAM PARAMETERS'       <=== VERIFY PARAMETERS
/**
/** INCLUDE THE TCP/API SOCKET INCLUDE (.H) DATA SET IN THE
/** COMPILER SYSLIB CONCATENATION.
/**
//COMPILE.SYSLIB DD DISP=SHR,DSN=TRGINDX.H <=== VERIFY DSNAME
//              DD DISP=SHR,DSN=&VSCCHD&CVER&EDCHDRS
/**
/** INCLUDE THE TCP/API SOCKET SUBROUTINE LIBRARY DATA SET
/** IN THE LINKAGE EDITOR SYSLIB CONCATENATION.
/**
//LKED.SYSLIB DD
//              DD
//              DD DISP=SHR,DSN=TRGINDX.CILIB <=== VERIFY DSNAME
/**
//LKED.SYSIN DD DUMMY,DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120)

```

Compile/Link IBM C/370 C Reentrant Program

This JCL is for the C/370 version of the compiler. If you are using the Ad/Cycle compiler, replace the following line (shown in bold in the JCL):

```

// DD DISP=SHR,DSN=&VSCCHD&CVER&EDCHDRS

```

with this line:

```

// DD DISP=SHR,DSN=&LNGPRFX..SEDCDHDR

//CLGIBMCR JOB
/**
/** SAMPLE JCL TO COMPILE, LINK, AND EXECUTE A REENTRANT
/** USER PROGRAM USING THE TCP/API C SOCKET LIBRARIES
/** AND THE IBM C/370 C COMPILER.
/**
/** EDIT THE JOB JCL STATEMENT, VERIFY THE DATA SET NAME(S)
/** OF THE USER'S DATA SETS, AND VERIFY THAT THE DATA SET
/** NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
/** FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED
/** ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB
/** ASSUMES THE STANDARD IBM C/370 EDCCPLG JCL PROCEDURE IS
/** AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
//
//CLGRENT EXEC EDCCPLG,
//      INFILE='USER.C(CPROG)',           <=== VERIFY DSNAME
//      CPARAM='RENT,DEF(IBM)',
//      GPARM='PROGRAM PARAMETERS'       <=== VERIFY PARAMETERS
/**
/** INCLUDE THE TCP/API SOCKET INCLUDE (.H) DATA SET IN THE
/** COMPILER SYSLIB CONCATENATION.
/**
//COMPILE.SYSLI DD DISP=SHR,DSN=TRGINDX.H <=== VERIFY DSNAME
//              DD DISP=SHR,DSN=&VSCCHD&CVER&EDCHDRS
/**
/** INCLUDE THE TCP/API SOCKET SUBROUTINE OBJECT LIBRARY
/** DATA SET IN THE PREPROCESSOR SYSLIB CONCATENATION.
/**
//PLKED.SYSLIB DD DISP=SHR,
//              DSN=TRGINDX.CIROBJ       <=== VERIFY DSNAME
//PLKED.SYSIN DD DSN=*.COMPILE.SYSLIN,DISP=(OLD,DELETE)

```

```

//          DD *
//      INCLUDE SYSLIB(S0SKCF)
//      INCLUDE SYSLIB(S0INTR)
//      ENTRY  CEESTART
//
//

```

Compile/Link SAS/C Non-reentrant Program

```

//CLGSASC JOB
//*
//* SAMPLE JCL TO COMPILE, LINK, AND EXECUTE A NONREENTRANT
//* USER PROGRAM USING THE TCP/API C SOCKET LIBRARIES
//* AND THE SAS/C C COMPILER. THIS SAMPLE WILL
//* WORK WITH SAS/C 4.50, 5.00, 5.01 AND 5.50C.
//*
//* EDIT THE JOB JCL STATEMENT, VERIFY THE DATA SET NAME(S)
//* OF THE USER'S DATA SETS, AND VERIFY THAT THE DATA SET
//* NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
//* FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED
//* ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB
//* ASSUMES THAT THE STANDARD SAS/C LC370CLG JCL PROCEDURE
//* IS AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
//*
//CLNORENT EXEC LC370CLG,
//  PARM.C='DEF(SASC),NORENT',
//  PARM.GO='PROGRAM PARAMETERS'                <=== VERIFY PARAMETERS
//*
//*      TCP/IP INCLUDE FILE DATA SET MUST PRECEDE SAS/C
//*      DATA SET.
//*
//C.SYSLIB DD DISP=SHR,DSN=TRGINDX.H            <=== VERIFY DSNAME
//          DD DISP=SHR,DSN=&MACLIB
//C.SYSIN  DD DISP=SHR,
//          DSN=USER.C(CPROG)                  <=== VERIFY DSNAME
//
//*      TCP/IP OBJECT SYSLIB DATA SET MUST PRECEDE SAS/C
//*      DATA SETS.
//*
//LKED.SYSLIB DD DISP=SHR,
//            DSN=TRGINDX.CSLIB<=== VERIFY DSNAME
//            DD DISP=SHR,DSN=SASC.&ENV.LIB    <=== VERIFY DSNAME
//            DD DISP=SHR,DSN=&SYSLIB
//            DD DISP=SHR,DSN=&CALLLIB
//LKED.SYSIN DD DUMMY,DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120)

```

Compile/Link SAS/C Reentrant Program

```

//CLGSASCR JOB
/**
/** SAMPLE JCL TO COMPILE, LINK, AND EXECUTE A REENTRANT C
/** PROGRAM USING THE TCP/API C SOCKET LIBRARIES AND THE
/** SAS/C C COMPILER. THIS SAMPLE WILL WORK WITH SAC/C 4.50,
/** 5.00, 5.01 AND 5.50C.
/**
/** EDIT THE JOB JCL STATEMENT, VERIFY THE DATA SET NAME(S)
/** OF THE USER'S DATA SETS, AND VERIFY THAT THE DATA SET
/** NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
/** FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED
/** ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB
/** ASSUMES THAT THE STANDARD SAS/C LC370C AND LC370LRG JCL
/** PROCEDURES ARE AVAILABLE IN YOUR INSTALLATION'S
/** PROCLIB(S).
/**
/** STEP 1: COMPILE USER PROGRAM REENTRANTLY.
/**
//CCRENT EXEC LC370C,
// PARM.C='RENT,DEF(SASC)'
/**
/** TCP/IP INCLUDE FILE DATA SET MUST PRECEDE SAS/C
/** DATA SET.
/**
//C.SYSLIB DD DISP=SHR,DSN=TRGINDX.H <=== VERIFY DSNAME
// DD DISP=SHR,DSN=&MACLIB
//C.SYSIN DD DISP=SHR,
// DSN=USER.C(CPROG) <=== VERIFY DSNAME
/**
/** STEP 2: LINK USER PROGRAM USING SAS/C CLINK
/** PREPROCESSOR AND THEN EXECUTE.
/**
//LKRENT EXEC LC370LRG,PARM.LKED='LIST,MAP,RENT',
// PARM.GO='PROGRAM PARAMETERS' <=== VERIFY PARAMETERS
/**
/** TCP/IP OBJECT SYSLIB DATA SET MUST PRECEDE SAS/C
/** DATA SETS.
/**
//LKED.SYSLIB DD DISP=SHR,DSN=TRGINDX.CSROBJ <=== VERIFY DSNAME
// DD DDNAME=AR#&ALLRES
// DD DISP=SHR,DSN=SASC.&ENV.OBJ <=== VERIFY DSNAME
// DD DISP=SHR,DSN=&SYSLIB
// DD DISP=SHR,DSN=&CALLLIB
//LKED.SYSIN DD DISP=(OLD,DELETE),DSN=*.CCRENT.C.SYSLIN
// DD *
// INCLUDE SYSLIB(S0SKCF)
// INCLUDE SYSLIB(S0INTR)
// ENTRY MAIN
/*
//

```

Customizing Socket Programs

Socket programs can be easily customized without the need to change and recompile the `sockcf` file. This leaves the `sockcf` object file untouched for use by other socket applications, but allows individual versatility for testing and customizing within the program itself. The fields of the `sockcfg` struct can be changed by including the header file `serrno.h` or `sockcfg.h` in your source program and setting the desired fields in the externally declared variable `s0skcfg`.

Note: You must have the current version of the C compiler installed to change the `sockcfg` values. If this environment is not available, contact your Customer Support specialist to investigate other options.

Refer to [Using Socket Libraries](#), which describes the fields of the `sockcfg` struct, before changing the default values.

Changing Values

These statements demonstrate how these values can be changed within the socket program:

```
#include <sockcfg.h>
.
. ( other include files )
.
main()
{
    s0skcfg.subsysid = "ABCD";           /* set subsysid */
    s0skcfg.apcboptc = APCBOTRC;       /* set api trace on */
    ( rest of program )
}
```

UNIX System Services MVS Integrated Sockets

This chapter contains information about Unicenter TCPAccess compatibility with UNIX System Services (formerly OpenEdition) MVS Integrated Sockets.

It includes the following sections:

- [References](#) – Lists reference material
- [Installation Considerations](#) – Describes installation steps specific for UNIX System Services socket support
- [Configuration Information](#) – Describes configuration changes specific to UNIX System Services socket support
- [Additional Socket Files](#) – Describes the additional files included in the SAMP data set
- [Additional Socket Call Parameters](#) – Describes options for `ioctl()`, `getsockopt()` and `setsockopt()`
- [Resolving Names and Addresses](#) – Describes how to do host name and address resolution for UNIX System Services sockets
- [TSO Commands](#) – Describes CONVXL8, LOADXL8 commands
- [Debugging Information](#) – Describes how to get control block information
- [Stopping and Starting Sockets](#) – Describes how to start and stop the socket API
- [Limitations](#) – Describes the limitations of UNIX System Services sockets
- [Release Information](#) – Describes support for UNIX System Services by MVS releases
- [Common INET Support](#) – Describes the use, advantages, and disadvantages of Common INET support

Unicenter TCPAccess sockets support the UNIX System Services Integrated Sockets as provided in IBM MVS/ESA. The first release of OpenEdition was part of MVS/ESA 4.3. OpenEdition socket support was introduced with MVS/ESA 5.1 and above. Review your IBM documentation to determine UNIX System Services support for your site.

References

This chapter provides information on Unicenter TCPAccess socket compatibility with UNIX System Services Integrated Sockets. It is not intended as a reference for IBM UNIX System Services MVS Integrated Sockets.

For more information on UNIX System Services sockets, check the following references:

AD/Cycle C/370 Library Reference: OpenEdition MVS Sockets (IBM Manual number SC23-3024-xx (MVS/ESA 5.1 version))

IEEE POSIX Standard 1003.1g, Protocol Independent Interfaces (ISO/IEC JTC 1/SC22/WG15N, P1003.1g/Draft 6.0 (January 1995), Information Technology - POSIX - Part xx: Protocol Independent Interfaces)

X/Open CAE Specification, Networking Services, Issue 4 (X/Open Document Number C438)

C/C++ for MVS/ESA V3.1 C/MVS Library Reference: OpenEdition MVS Sockets (IBM Manual number SC23-3875-xx)

MVS/ESA Application Development Reference: Assembler Callable Services for OpenEdition MVS (IBM manual number SC23-3020-01 (MVS/ESA 5.2 version))

MVS/ESA Application Development Reference: Assembler Callable Services for OpenEdition MVS (IBM manual number SC23-3020-02 (MVS/ESA 5.2.2 version))

MVS/ESA OpenEdition MVS File System Interface Reference (IBM manual number SC23-3802-00 (MVS/ESA 5.1 and 5.2 versions))

MVS/ESA OpenEdition MVS File System Interface Reference (IBM manual number SC23-3802-01 (MVS/ESA 5.2.2 version))

Installation Considerations

The UNIX System Services Physical File System (PFS) and the Socket API are both installed automatically.

UNIX System Services PFS Transport Driver load modules are installed in the PFSLOAD partitioned data set. This data set must be APF authorized and must be in the STEPLIB concatenation for UNIX System Services MVS or be part of the system link list concatenation.

Important! *The Unicenter TCPaccess address space must be defined to your security package (such as RACF, ACF2, and so on) with a valid UNIX System Services MVS security segment.*

The UNIX System Services MVS procedure must be stopped and restarted after Unicenter TCPaccess is installed and after the PFSLOAD data set is added to the UNIX System Services MVS PROC or the system STEPLIB. UNIX System Services must be configured prior to the restart.

Configuration Information

Configuration information for Unicenter TCPaccess UNIX System Services Integrated sockets is presented in the *Unicenter TCPaccess Communications Server Customization Guide*.

Additional Socket Files

Included in the SAMP data set, there are two members needed for UNIX System Services socket support. The header include file icssckt.h includes defines for UNIX System Services socket function specific to the TCPaccess socket implementation. There is also an equivalent assembler macro, ICSSCKTM.

You will need to move these files to your own libraries to use them.

Important! *Use of definitions in these files may make socket applications binary incompatible with the IBM TCP/IP or IBM Any/Net socket implementations.*

Additional Socket Call Parameters

In addition to the socket call parameters described in the chapter “Library Functions,” the UNIX System Services integrated sockets support additional parameters for several socket calls.

ioctl() Parameters

The following additional parameters for the `ioctl()` call are supported for UNIX System Services integrated sockets users.

<code>FIOGETOWN_Ref328468229</code>	Gets the process or process group ID specified to receive signals. This value is type <code>int</code> .
<code>FIOSETOWN</code>	Sets the process or process group ID specified to receive signals. This value is type <code>int</code> .
<code>SIOCADDRT2</code>	Adds a Unicenter TCPaccess format route entry.
<code>SIOCDELRT2</code>	Deletes a Unicenter TCPaccess format route entry.
<code>SIOGIFHWADDR</code>	Gets the hardware address. Uses <code>if_req</code> struct.
<code>SIOCGIFNUM</code>	Gets the number of interfaces. Uses the <code>if_req</code> struct.
<code>SIOCGIFMTU</code>	Gets the interface MTU (Maximum Transmission Unit). Uses the <code>if_req</code> struct.
<code>SIOCGPGRP</code>	Synonym for <code>FIOGETOWN</code> . This value is type <code>int</code> . This parameter is defined in UNIX System Services MVS 5.2.2.
<code>SIOCSIFMETRIC</code>	Sets the network interface routing metric. This value is type <code>int</code> . Uses the <code>if_req</code> struct.
<code>SIOCSPPGRP</code>	Synonym for <code>FIOSETOWN</code> . This value is type <code>int</code> . This parameter is defined in UNIX System Services MVS 5.2.2.

Note: The `SECIGET` command is not supported since it is valid only in the `AF_UNIX` domain.

The `FIOASYNC` command documented in 1003.1g is not supported. This command enables signal-driven I/O and cannot be easily implemented until it is supported by UNIX System Services sockets.

getsockopt() Parameters

The following additional parameters for the `getsockopt ()` call are supported for UNIX System Services integrated sockets users.

Socket-Level Options

Socket-Level options for `getsockopt()`:

<code>SO_ACCEPTCONN</code>	Reports whether socket listening is enabled (that is, <code>listen()</code> was issued).
<code>SO_ERROR</code>	Reports information about error status and clears it.
<code>SO_KEEPALIVE</code>	Reports whether connections are to be kept open with periodic transmissions of messages. Use <code>TCP_KEEPALIVE</code> to report the current interval between packets.
<code>SO_RCVLOWAT</code>	Reports minimum number of bytes to process for socket input operations.
<code>SO_RCVTIMEO</code>	Reports timeout value for socket input operations. Note: The timeout value is valid only for MVS 5.2.2 and above.
<code>SO_REUSEADDR</code>	Reports whether the rules used in validating addresses supplied to <code>bind()</code> should allow reuse of local addresses.
<code>SO_SNDLOWAT</code>	Reports minimum number of bytes to process for socket output operations.
<code>SO_SNDTIMEO</code>	Reports timeout for output operations.
<code>SO_TYPE</code>	Reports socket type.

TCP-Level Options

TCP-Level options for `getsockopt()`:

<code>TCP_KEEPALIVE</code>	Reports the current time interval (in seconds) between keepalive packets. This parameter is given as type <code>int</code> . The option <code>SO_KEEPALIVE</code> must be enabled first.
----------------------------	--

UDP-Level Options

UDP-Level options for `getsockopt()`:

`UDP_CHECKSUM` Reports whether UDP checksum computation is to be performed. This parameter is given in type `int`.

IP-Level Options

IP-Level options for `getsockopt()`:

`IP_HDRINCL` Reports whether, for a `SOCK_RAW` socket, the complete IP header is included with the data on send operations.

`IP_OPTIONS` Reports whether IP options are to be transmitted in the IP header of each outgoing packet and what those options are.

`IP_TOS` Reports the type-of-service field in IP header of outgoing packets.

`IP_TTL` Reports the time-to-live field in IP header of outgoing packets.

`setsockopt()` Parameters

The following additional parameters for the `setsockopt()` call are supported for UNIX System Services integrated Sockets users.

Socket-Level Options

Socket -Level options for UNIX System Services `setsockopt()`:

SO_KEEPALIVE	Sets whether connections are to be kept open with periodic transmissions of messages. Packet interval is the TIB <code>KEEPALIVETIMER</code> value. Use <code>TCP_KEEPALIVE</code> to change the interval.
SO_RCVLOWAT	Sets minimum number of bytes to process for socket input operations. You can set this option but it will be ignored for <code>SOCK_DGRAM</code> and <code>SOCK_RAW</code> socket types.
SO_REUSEADDR	Sets whether the rules used in validating addresses supplied to <code>bind()</code> should allow reuse of local addresses. When enabled, local addresses already in use may be bound. The system checks at connect time to ensure that the set <code><laddr, lport, raddr, rport></code> are not already in use by another association. If the set is already in use, <code>EADDRINUSE</code> is returned. The system checks at listen time to see if the port is already being listened on.
SO_SNDLOWAT	Sets minimum number of bytes to process for socket output operations. You can set this option but it will be ignored for <code>SOCK_DGRAM</code> and <code>SOCK_RAW</code> socket types. Note: <code>SO_RCVBYTCNT</code> , <code>SO_RCVREQCNT</code> , <code>SO_SNDBYTCNT</code> , <code>SO_SNDREQCNT</code> , <code>SO_SENDALL</code> , and <code>SO_READFRAG</code> are not supported for UNIX System Services sockets.

TCP-Level Options

TCP-Level options for `setsockopt()`:

TCP_KEEPALIVE	Sets the time interval between keepalive packets in seconds. <code>SO_KEEPALIVE</code> must be enabled first.
---------------	--

UDP-Level Options

UDP-Level options for `setsockopt()`:

UDP_CHECKSUM	Sets whether UDP checksum computation is to be performed. This value is given in type <code>int</code> . 0 = OFF, non-zero = ON.
--------------	---

IP-Level Options

IP-Level options for `setsockopt()`:

<code>IP_HDRINCL</code>	Sets whether, for a <code>SOCK_RAW</code> socket, the complete IP header is included with the data on send operations.
<code>IP_OPTIONS</code>	Sets whether IP options are transmitted in the IP header of each outgoing packet and what those options are.
<code>IP_TOS</code>	Sets the type-of-service field in IP header of outgoing packets.
<code>IP_TTL</code>	Sets the time-to-live field in IP header of outgoing packets.

`recvmsg()`

Ancillary data will not be supported in this release. Ancillary data is referenced via the `msg_hdr` fields `msg_control` and `msg_controllen`. These fields are not yet supported in the UNIX System Services `msg_hdr` structure.

Resolving Names and Addresses

UNIX System Services MVS version 1.2 uses the LE/370 version 1.3 or 1.4 runtime libraries to perform certain socket related functions such as `gethostbyname()`, `getprotobyname()`, and so forth. To perform this functionality, the LE/370 runtime library (RTL) reads specific MVS data sets to map services to names and to obtain domain name resolution configuration information.

Note: These data sets and the utilities to build them are NOT distributed with either UNIX System Services MVS or the LE/370 Runtime Library.

IBM distributes these data sets with the IBM TCP/IP product for MVS only.

This section describes the steps necessary to build and configure these required data sets using sample members and utilities provided with the Unicenter TCPaccess product.

The LE/370 version 1.5 RTL uses members in the `/etc` directory for the same functionality as the MVS data sets.

Using /etc Files with UNIX System Services for Domain Name Resolution

UNIX System Services users running MVS/ESA 5.2.1 or earlier are limited to the MVS host files described in this chapter for resolving host name and other DNR requests.

With MVS/ESA 5.2.2, it is possible to use HFS /etc files for DNR resolution.

To use the /etc files, the following program changes are necessary.

1. Delete the `#include` for `<manifest.h>`.
2. Replace any `#include` of `<bsdtypes.h>` with `<sys/types.h>`.
3. Replace any `#include` of `<bsdtime.h>` with `<sys/time.h>`.
4. Remove `SYS1.SFOMHDRS` from the `-INC` list for C89 or any compile PROCLIB cataloged procedures.
5. Replace any `#define _OPEN_SOCKETS` (or other references to `_OPEN_SOCKETS`) with `_OE_SOCKETS`.
6. If you are using threads, replace any `extern h_errno` with `#define h_errno *(__herrno())`.
7. Where RPC functions are being used, add a `#include` for `<rpc/netdb.h>`.

Making these changes generates the code needed to reference the /etc files for DNR requests. If no changes are made, the MVS files are used as described in the rest of this chapter.

Data Sets for Host Resolution

The LE/370 RTL version 1.3 and 1.4 require five data sets to correctly perform the supporting socket functions such as `gethostbyname()`, `getprotobyname()`, and so on.

These five data sets are:

<code>prefix.TCPIP.DATA</code>	Master configuration file.
<code>prefix.ETC.PROTO</code>	Protocol name mappings.
<code>prefix.ETC.RPC</code>	RPC service name mappings.
<code>prefix.ETC.SERVICES</code>	Service name mappings.
<code>prefix.STANDARD.TCPXLBIN</code>	ASCII to EBCDIC translation file.

Two other data sets are used by the RTL if host name resolution is being done in local mode. These data sets are built with the IBM `MAKESITE` utility. Unicenter `TCPaccess` does *not* provide a replacement for `MAKESITE` because the RTL can perform host name resolution using name servers without these other data sets, and users without a name server can upgrade to LE/370 version 1.5 and configure hosts in `/etc/hosts` as a workaround.

The two other data sets are:

```
prefix.SITEINFO
prefix.ADDRINFO
```

The `prefix.TCPIP.DATA` data set is the primary configuration member for the RTL. It defines the prefix for the rest of the data sets used by the RTL and its prefix may be different from those data sets. In addition, it provides the configuration information for host name resolution process (for example, 148.52.128.104) and resolver settings such as protocol (UDP), port (53), timeout (30 seconds) and retries (3).

Search Procedures

The RTL performs a complex search sequence when it needs to locate the prefix.TCPIP.DATA data set, since there is no generic way to define to the RTL what the prefix actually is.

The RTL attempts to locate the data set using the following search sequence:

1. If the environment variable, RESOLVER_CONFIG is defined, it uses this value to locate the data set. Usually this variable would be set in each user's profile member in their home directory.

```
export RESOLVER_CONFIG="//'"IOS.OMVS.TCPIP.DATA'"
```

Note: The name must have two leading forward slashes and the actual data set name must be enclosed by a double-quote:single-quote:double-quote sequence (" / ") on each end of the name.

2. It then looks for a SYSTCPDD DD statement. This is not recommended for any application which fork(s) due to UNIX System Services MVS's failure to copy data set allocations across a fork.

```
//SYSTCPDD DD DSN=IOS.OMVS.TCPIP.DATA,DISP=SHR
```

3. The jobname or userid is used as the prefix.

For user IBMUSER, the RTL searches for IBMUSER.TCPIP.DATA.

4. The RTL then looks for SYS1.TCPPARM(TCPDATA).
5. The RTL then looks for TCPIP.TCPIP.DATA.

Configuration

Users migrating from IBM's TCP/IP for MVS to Unicenter TCPAccess should note that the IBM TCP/IP installation utility EZAPPRFX does not affect the default prefix used by the Run Time Library (for example, TCPIP).

The DATASETPREFIX must be configured in the *prefix*.TCPIP.DATA data set. It must be updated to specify the prefix used by the other four data sets. The DOMAINORIGIN keyword must be configured with the site's domain name (for example, hq.company.com). NSINTERADDR keyword must be configured with the sites domain name server. You can update the other keywords if desired to tune the resolver process.

The data sets, *prefix*.TCPIP.DATA, *prefix*.ETC.PROTO, *prefix*.ETC.RPC and *prefix*.ETC.SERVICES, must be allocated with attributes of RECFM=FB, LRECL=80, BLKSIZE=3120 and with an allocation of one track. You should copy sample members TCPDATA, ETCPROTO, ETCRPC and ETCSEV, respectfully, from the IOS390.SAMP data set to the new data sets after allocation.

The *prefix*.STANDARD.TCPXLBIN data set must be allocated with attributes of RECFM=FB, LRECL=256, BLKSIZE=256 and an allocation of one track. Its purpose is to provide ASCII-to-EBCDIC translation tables for the domain name resolution process. It is created using either the CONVXL8 or the LOADXL8 utilities.

```
LOADXL8 ENGLISH 'IOS390.OMVS.STANDARD.TCPXLBIN'
```

Users of the LE/370 RTL Version 1.5 and above must configure members in the /etc directory instead of in MVS data sets. The /etc/protocols, /etc/rpc and /etc/services can be created using the OPUT command and the Unicenter TCPAccess sample members.

```
OPUT 'IOS390.SAMP(ETCPROTO)' '/etc/protocols'.
```

LE/370 RTL Version 1.5 and above can require the configuration of other /etc members such as, /etc/resolv.config, /etc/hosts and /etc/networks.

A sample /etc/resolv.config member is:

```
domain hq.company.com
search hq.company.com ohio.company.com company.com
```

```
nameserver 148.52.32.165
```

```
nameserver 148.52.32.56
```

A sample /etc/hosts member is:

```
127.0.0.1      localhost localhost
148.52.32.165  homehost homehost.md.mycompany.com
```

A sample `/etc/networks` member is:

```
127      loopback
148.52   mycompany
```

Consult IBM C/C++ for MVS/ESA, C/MVS Library Reference: *OpenEdition MVS Sockets*, Version 3 Release 1, Document Number SC23-3875-00, Program Number 5655-121, Section: 1.3.6.1, "Understanding TCP/IP Data Set Names with OpenEdition MVS" for more information on configuring UNIX System Services (OpenEdition) MVS and the LE/370 RTL.

TSO Commands

This section describes the TSO commands available to Unicenter TCPaccess UNIX System Services socket users.

CONVXL8 Command

The TSO command CONVXL8 converts a table from editable text to binary.

CONVXL8 creates a data set with three records. Each record is 256 bytes in length.

- The first record has "TCP/IP translate tables" (in EBCDIC) starting in column one, with the remainder of the record padded with EBCDIC blanks (X'40').
- The second record has 256 EBCDIC values representing the ASCII-to-EBCDIC translation.
- The third record has 256 ASCII values representing the EBCDIC-to-ASCII translation.

File names use TSO prefix as defined by TSO rules. A fully-qualified data set name needs to be enclosed in quotes. A data set name without quotes can have a user-specified prefix placed before the name. This prefix is defined by the user's TSO profile. Refer to TSO documentation for more information about prefixes.

```
CONVXL8 INPUT OUTPUT
```

INPUT	<p>Required. Specifies the source data set to be converted.</p> <p>The data set must be in standard IBM format for SBCS translation tables. If input is a PDS member, INPUT should be specified as <i>dsname(member)</i>.</p> <p>Default: None.</p> <p>Note: Translate tables in the SAMP data set are not in this format. Use the TSO command LOADXL8 to prepare them for use with UNIX System Services. See LOADXL8 Command for more information.</p>
OUTPUT	<p>Required. Specifies the output data set created by the conversion.</p> <p>If output is a PDS member, OUTPUT should be specified as <i>dsname(member)</i>.</p> <p>Default: None.</p> <pre>CONVXL8 LIB.SOURCE(TRANS) LIB.TRANTAB</pre> <p>This reads USER.LIB.SOURCE(TRANS) and creates a translate table in USER.LIB.TRANTAB.</p> <pre>CONVXL8 'SYSTEM.TCP.DATA(TRAN)' 'SYSTEM.BIN.TRANS'</pre> <p>This reads SYSTEM.TCP.DATA(TRAN) and creates a translate table in SYSTEM.BIN.TRANS.</p>

LOADXL8 Command

LOADXL8 has the same functionality as the CONVXL8 command, but it reads the load module from compiled translate tables as input. Note that it does not read the source. It loads the module from STEPLIB or TSO TASKLIB. The module can be converted for UNIX System Services use with this command.

```
CALL 'trgindx.LOAD(LOADXL8)' 'table-name' 'indx.STANDARD.TCPXLBIN' ' '
```

<i>trgindx</i>	Unicenter TCPAccess high-level qualifier.
<i>table-name</i>	Load module name of translate table to be loaded.
<i>indx</i>	UNIX System Services high-level qualifier.

ILATCH command

An IFS command that allows you to display and free latches used to serialize data.

For more information about the ILATCH command, see the *System Management Guide*.

Debugging Information

This section describes procedures to help you debug problems with Unicenter TCPaccess UNIX System Services support.

Initialization

If you are having problems with the initialization of UNIX System Services sockets, check the following:

- Examine the SMP/E job output to verify successful processing.
- Verify the release and maintenance levels of UNIX System Services - Version 5.1, 5.2, or 5.2.2.
- Verify the PFS configuration statements:
 - FILESYSTYPE
 - NETWORK
 - UNIX System Services MVS PROC
- Examine any IVP return codes or other return codes or messages from UNIX System Services utilities.
- Verify the logs for error messages. Also, look at the console log and any dump data sets for abends that may have been created.
- Verify non-UNIX System Services access to Unicenter TCPaccess. Use tools such as ping, FTP, or Telnet to verify that the Unicenter TCPaccess transport provider is functional.

Application Issues

If you are having problems with a UNIX System Services application, check the following:

- Before calling Customer Support, get a good description of the failure with symptoms, return code, errno, and errno junior codes. Examine any messages from the application. Be sure to include a brief description of the application and known socket services.
- Verify the release and maintenance level of UNIX System Services/OpenEdition – Version 5.1, 5.2, or 5.2.2.
- Examine the logs for error message. Look at the console log and any dump data sets for abends that may have been created.
- Isolate the problem to your application by testing the status of other UNIX System Services applications and non-UNIX System Services application functionality (ping, Telnet, FTP, and so on).
- Issue a “TCP SNAP ALL” command immediately after any application failure.

TCP SNAP ALL Command

The existing command:

```
/F IOS390,TCP SNAP ALL
```

was modified to dump the UNIX System Services PFS Transport Provider and Socket API control blocks in the Unicenter TCPaccess address space.

Stopping and Starting Sockets

The socket API can be stopped and restarted by stopping and restarting the ACP task group within Unicenter TCPAccess.

Note: The UNIX System Services MVS PFS can only be stopped and restarted by stopping and restarting UNIX System Services MVS.

Limitations

The number of Socket API endpoints supported by the UNIX System Services PFS and Socket API is currently limited by the following constraints:

- The amount of virtual storage in the Unicenter TCPAccess address space.
- The number of ports. Socket API programs generally does not share port numbers.

The number of port numbers is limited to 65,535 for both TCP and UDP. Client applications that reuse the same port number when connected to different servers can help ease this limitation.

The range of ports for both TCP and UDP can be decreased by setting overriding values on the TCP and UDP statements of TCPCFGxx. Setting these values can decrease the number of ports and sockets.

- UNIX System Services MVS requires that a maximum socket number be set.
- One latch per endpoint, with a maximum of 32,767.

Release Information

Unicenter TCPAccess uses operating system facilities and architecture dependent instructions that are only available in MVS/ESA versions 3.1.3 and above.

The UNIX System Services PFS only supports MVS/ESA version 5.1.0 and above.

The UNIX System Services PFS Pre-Router is only supported on MVS/ESA 5.2.2 and above.

Common INET Support

IBM offers an UNIX System Services MVS facility within MVS/ESA Version 5.2.2 and higher that allows a socket program to be used with multiple TCP/IP stacks simultaneously, without knowledge of the application and without coding modifications.

Note: Although this may appear to be very beneficial, use of this feature is not without risk and customers are strongly advised to consider alternatives before implementing Common Inet Support.

Benefits of INET Support

The benefits of using Common Inet Support are:

- Applications are developed as if they were using the AF_INET family with a single TCP/IP stack
- The system administrator adds and removes TCP/IP address spaces to the system configuration as required without application knowledge
- Applications, if coded properly, may choose a particular TCP/IP region by issuing an `ioctl()` call after the socket is created

Problems of INET Support

The problems with using Common Inet Support are:

- Multiple copies of TCP/IP running on the host all use the AF_INET value of two in order to appear to be a single Physical File System. This presents the following problems:
 - Host resource consumption is higher than that single region or multiple regions using different address family values. This is mainly the result of the Common Inet Layer opening a socket to every Physical File System every time a socket function is issued.
 - Other socket function calls up through the time a connection is established via a particular Physical File System must be processed by all Physical File Systems.
 - Once an outbound connection is established over a single Physical File System, the sockets created within the other Physical File Systems are closed by the Common Inet Layer. Listening sockets and datagram type sockets are always open in all Physical File Systems.
- There are no conflicting parameters to deal with such as interface name, local host IP address, and interface addresses. Loopback connections are made via the selected Physical File System.

Problems of Multiple Physical File Systems

The problems with using multiple Physical File Systems are:

- If an application requires the use of more than one TCP/IP region at the same time, the application must open a socket to each. This, however, is beneficial if one TCP/IP region is recycled. The application can close the socket to that region, and then reopen it when the region is restarted. There is no bad side effect to the other sockets connected to other TCP/IP regions. On the positive side of this, however, the application immediately knows of a failure of one region.
- The application will most likely need to be using non-blocking socket functions. This is because events can occur simultaneously on multiple Physical File Systems (for example, two inbound connects occur at the same time, one on each TCP/IP region).

Established connections are not fault tolerant across TCP/IP regions (only within a TCP/IP region).

SAS/C Socket Library Interface (LSCNCOM)

This appendix describes the LSCNCOM interface between the SAS/C socket library and the API.

SAS/C Socket Interface

The SAS/C socket interface, LSCNCOM, is a vendor independent socket library that is provided as part of the SAS/C compiler. This interface allows programs to be written that can use different vendors' C socket interface at execution time without having to relink in vendor dependent code.

The LSCNCOM interface relies on the SAS/C compiler and runtime library at the version 5.50 level or higher.

SAS C TCPIP sockets has the following features:

- Uses no H files supplied with this Unicenter TCPaccess C interface.
- Uses no RPC support libraries supplied with this Unicenter TCPaccess C interface.
- C socket calls are processed by routine LSCNCOM.

The following are the advantages of having a program run over SAS C TCP/IP sockets (LSCNCOM):

- The C socket support routines are loaded at runtime, eliminating the need to link C socket support routines into the application.
- Only SAS routines are used during the compile, prelink and linkedit – reducing the likelihood of vendor conflicts during compile, prelink, linkedit and runtime events.
- Updates to the C socket support routines are placed in load module LSCNCOM. Therefore, changes to C socket support routines do not require relinking to an application to accrue the change benefits when C support routines are modified. Applying a PTF to C socket support routines changes LSCNCOM and does not require the application programmer to know they must relink their application as with TLI C sockets.

There are two versions of LSCNCOM that Unicenter TCPaccess can use:

- LSCNCOM provided by CA in the SASLINK library.
- SAS also provides a different but functionally equivalent version of LSCNCOM that we can use through our HPNS, IUCV and OE interfaces.

SAS's LSCNCOM can talk to either Unicenter TCPaccess' or IBM's TCP/IP implementation using any of the OE, HPNS, and IUCV API interfaces.

To use SAS's LSCNCOM with the HPNS interface ensure that the STEPLIB contain:

- SAS's LSCNCOM ahead of Unicenter TCPaccess's LSCNCOM
- Unicenter TCPaccess's EZASOH03 ahead of IBM's EZASOH03

To use SAS's LSCNCOM with the IUCV interface ensure that the STEPLIB contain:

- SAS's LSCNCOM ahead of Unicenter TCPaccess' LSCNCOM
- Unicenter TCPaccess's EZASOK03 ahead of IBM's EZASOK03

When the application has not requested a specific API interface SAS's LSCNCOM attempts to connect using APIs in the following order:

- OE
- HPNS
- IUCV

SAS's LSCNCOM decides which API interface to use when not directed by the application. It makes this decision independently on the first SAS C TCP/IP socket call on every TCB. The first SAS C TCP/IP C socket related call on a TCB is processed using its internal logic even if a higher-level TCB has already opened up an HPNS session. Now the application can direct each SAS TCB environment via putenv() calls with the appropriate variables. SAS does not propagate the EXEC parm overrides to any daughter task. One cannot set '=TCPIP_OE=1' or '=TCPIP_HPNS=1' in the EXEC PARM statement and propagate it across TCB levels. If the top job task is not a SAS task then the EXEC parm overrides are not used at all for any of the SAS TCBs.

The TCPIP-MACH variable is used differently by each version of LSCNCOM:

- The TCPIP_MACH variable when running with SAS's LSCNCOM refers to the Unicenter TCPaccess job name
- The TCPIP_MACH variable when running with CA's LSCNCOM refers to the Unicenter TCPaccess subsystem name

Use caution when using the IUCV VMCF or HPNS interface. If the SYSTCPD DD is defaults and does not explicitly direct you to a specific TCP/IP interface you may end up using an unexpected TCP/IP interface.

Vendor notes on SAS/C TCP/IP socket programming:

Sites running SAS C TCP/IP socket applications with multiple tasks occasionally have instances where the givesocket()/takesocket() logic works for one TCP/IP implementation (Unicenter TCPaccess' or IBM's) of LSCNCOM but not the other.

To get both the IBM and Unicenter TCPaccess TCP/IP implementations to run (without any knowledge of the TCP/IP implementation) you should place a select() call after the givesocket() call waiting for the exception status on the passed socket. When the daughter task issues a takesocket(), it causes the select() on the mother task to return with the exception bit set for the passed socket. At this point, the mother task (givesocket()) should issue a close() for the socket.

Restrictions

The following features of the interface, as documented in the *SAS Technical Report C-111* (hereafter referred to as simply *C-111*), are not supported by this release of the Unicenter TCPaccess module:

- No support is provided for raw sockets or options that deal with the basic IP data stream. This includes the MSG_DONTROUTE option, ioctl() SIOCGIFxxx options, and setsockopt() SO_RAW option. In addition, other operations that refer to fields in the if.h structure may not be supported.
Note: See the *SAS Usage Note 1108* for a workaround zap that lets the RPC library function without raw support.
- SO_KEEPALIVE is not supported.
- Options F_GETFD and F_SETFD of fcntl() are not supported in the LSCNCOM routine since the #define values are being used for TCPaccess Unicenter options. This conflict will be resolved in a future release.
- The ioctl() option FIONREAD is not documented properly. The SAS manual states that this option returns a value of 1 if there is data to be read, and a VALUE of 0 if there is no data. However, this function actually returns the number of bytes waiting (if greater than zero), as the socket library (and the BSD man page) states.
- Writes that are flow-controlled may not be redriven.

- Only the `gethostbyname()` and `gethostbyaddr()` functions are handled by the Domain Name Resolver (DNR) when it is chosen. The database-related calls listed below (and described in *C-111*) are handled by the SAS resolver even though an equivalent file (and function) may be provided by Unicenter TCPaccess.

Therefore, to use all functions of the interface, the IBM-style `/etc` files (described in *C-111*) must also be defined.

<code>getnetbyname</code>	<code>getservbyport</code>	<code>setprotoent</code>	<code>getnetbyaddr</code>
<code>getpeerent</code>	<code>setservent</code>	<code>gethostbyname</code>	<code>getnetent</code>
<code>endpeerent</code>	<code>gethostbyaddr</code>	<code>getprotoent</code>	<code>endnetent</code>
<code>getprotobyname</code>	<code>getservent</code>	<code>endprotoent</code>	<code>getprotobyname</code>
<code>setpeerent</code>	<code>endservent</code>	<code>getservbyname</code>	<code>setnetent</code>

See the table under [Setup for SAS Socket /etc Files](#) for equivalent DNR parameter files.

- Due to Unicenter TCPaccess restrictions, the functions `givesocket()` and `takesocket()` are limited to passing sockets between tasks in the same address space.

Requirements

SAS/CONNECT requires a PTF from SAS in order to function correctly with the LSCNCOM routine. With this co-requisite fix, SAS validates the use of SAS/CONNECT with Unicenter TCPaccess and LSCNCOM.

Customers can run SAS/CONNECT or SAS/SHARE with Unicenter TCPaccess. In addition to the necessary Unicenter TCPaccess maintenance for the SAS/C 5.50 socket library, customers also need the following:

- V6.08 of SAS (the latest SAS major release)
AND
- Maintenance Level TS410 with the zap documented in SAS Note V6-SYS.SYS-08338
OR
- Maintenance level TS415 and no zap needed

Certification

The RPC portion of the SAS/CSL product has not been certified. Certification will be completed in a future release.

Note: SAS has a workaround zap, documented in their *Usage Note 1108*, that can be used to bypass some of the unsupported `ioctl()` options. The standard SUN/RPC `getmyaddress` function does not work with Unicenter TCPaccess because the `ioctl()` function supplied does not support `SIOCGIFCONF` or `SIOCGIFFLAGS` `ioctl` commands. Instead, a hard-coded loopback IP address (127.0.0.1) must be returned. To correct the problem, apply the zap referenced by Z1001108.

Usage

The `LSCNCOM` routine, and its alias `L$CNCOM`, should replace or be placed before the SAS-supplied version in the link-list search order. The routine is dynamically loaded on the first call to a SAS Socket Library function.

Using the Unicenter TCPaccess Variables

If you want to use the Unicenter TCPaccess socket variables, you must define the `Interlink` symbol in your source file. It must be placed before the `#include` statement.

Use the following as a guide:

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

Environment Variables

The environment variables below are recognized by the LSCNCOM interface:

ICS_SUBSYS The subsystem name of the Unicenter TCPaccess API task that was defined in the ACPCONxx parameter member. For compatibility with earlier releases, SUBSYS and TCPIP_MACH (first four bytes only) are also recognized.

Default: ACSS.

ICS_RESOLVER Defines the order in which the DNR and SAS resolver are used:

ONLY Unicenter TCPaccess resolver only, return OK or error.

FIRST Unicenter TCPaccess is called first; SAS is called if there is an error.

LAST SAS is called first; Unicenter TCPaccess is called if there is an error.

NEVER SAS resolver always, return OK or error.

These variables should be set prior to the first call to an LSCNCOM function. Either the PUTENV TSO command or inline (execute-time) parameters can be used to define or override the above variables.

Since it is difficult to delete permanent environment variables, the LSCNCOM interface treats a variable that is defined but has a null value as if it were not set. If none of the options are set, the default is used.

Default: ONLY

Set Up for SAS Socket /etc Files

The following table shows the configuration files used by the LSCNCOM interface, along with format documentation and equivalent parameter member for the DNR configuration. Either the files can be created with the expected names or environment variables can be setup to override the file names as documented in C-111.

SAS/C Default MVS Names	UNIX File	MAN Page	Equivalent DNR PARM Member
id.ETC.PROTOCOLS	/etc/protocols	protocols(4)	DNRPRTxx
id.ETC.SERVICES	/etc/services	services(4)	Syntax differs; you must create this from the DNRSVCxx member or copy from the workstation. The SAS resolver recognizes case-sensitive names, but DNRSLCxx is uppercase only.
id.ETC.HOSTS	/etc/hosts	hosts(4)	Syntax differs; you must create this from the DNRHSTxx member (static) or use the ICS or SAS resolver.
id.ETC.NETWORKS	/etc/networks	networks(4)	DNRNETxx DNRNETxx should be unnumbered or the SAS resolver interprets the sequence field as an alias.
id.ETC.RESOLV.CONF	/etc/resolv.conf	resolv.conf(4)	No equivalent; you must extract information from DNRSVCxx (domain name) and DNRNSCxx (name servers) to create this file. If no name servers are defined, static name resolution is used. See C-111.
id.ETC.RPC	/etc/rpc	rpc(4)	DNRRPCxx The overrides can either be via a DD name or via a fully qualified data set name.

Index

#

#include files, 1-2, 2-2, 4-1

A

ABEND, 3-14

abortive disconnect, 3-15

accept(), 3-8, 3-24

ACLOSE macro instruction, 1-7

acs.h, 3-19, 4-1

address, 3-60

address families, 4-3

address spaces, 3-15

AF_INET, 3-3, 3-7, 3-16, 3-27

ANSI C prototype statements, 1-3

AOPEN macro instruction, 1-8

APCB

- C struct correspondence, 2-1

- closing, 1-7

- compiler differences, 1-18

- initializing, 1-8

apcb struct, 2-1, 2-2

APCBASM, 1-8, 1-18

apcbenvr, 1-18

APCBSASC, 1-8, 1-18

APCBXL, C struct correspondence, 2-1

apcbxl struct, 2-1, 2-4

apclose(), 1-3, 1-7, 1-8

API subsystem termination, 1-21

api.h, 1-2, 2-2

api_end_exit(), 1-21

apopen(), 1-3

apopen()C library, functions, apopen(), 1-7

Application Program Control Block (APCB), 2-2

Application Program Control Block Exit List, 2-4

ASCII, 3-60, 3-61

assembler language, 1-2, 1-5, 1-18

association, 3-9, 3-31, 3-80, 3-106

ATCB, use with topen(), 1-15

atexit(), 3-106

ATTACH, 3-15

B

Berkeley Software Distribution (BSD), 3-2

bind(), 3-7

binding a protocol address, 1-12

blocking mode, 3-14

BSD, Berkeley Software Distribution, 3-2

buffering limits, 6-13

C

C library

- comparisons with assembler, 1-5

- functions

- apclose(), 1-7, 1-8
 - tcheck(), 1-9, 1-22
 - tclose(), 1-10
 - terror(), 1-11, 1-14, 1-17
 - texec(), 1-12
 - tferror(), 1-14
 - topen(), 1-15
 - tstate(), 1-16
 - twto(), 1-17

- installation, 6-2

C structs, correspondence to dsects, 2-1

cdefs.h, 4-1

cfuns struct, 4-3

CLGIBMC, 6-6, 6-18

CLGIBMCR, 6-6, 6-19

CLGSASC, 6-6, 6-20

CLGSASCR, 6-6, 6-21

close(), 3-13, 3-106, 6-13

closepass(), 3-18, 3-69

Common Inet Support, 7-18

communications domain

- description of, 3-3

- format of, 3-7

compilers

- C runtime library, 1-18, 3-22

- configuration files use with, 6-7

- exits using C library, 1-2

- I/O routines, 3-21

- interpretation of statements, 1-2

- JCL, 6-6

- socket configuration file, 6-14

- supported, 1-2

- urgent data handling, 3-17

configuration

- error codes, 4-3

- parameters, 4-3

- sockets, 6-6, 6-7

connect(), 3-9

connections,

- accepting, 3-8, 3-24, 3-64

- connect confirm indication, 1-20

- connect request indication, 1-20

- disconnect indication, 1-20

- initiating, 3-8

- listening for, 3-64

- shutting down, 3-13

- terminating, 3-27

constants, 1-2

CONVXL8 command, 7-13

customization

- C library, 6-7

- socket files, 6-22

D

DARPA Internet Domain, AF_INET, 3-3

data

- buffer manipulation, 6-13

- connected sockets, 3-10

- contiguous, 3-12

- duplicated, 3-106

- expedited data indication, 1-20

- input from socket, 3-75

- lost, 3-106

- noncontiguous, 3-12, 3-77

- normal data indication, 1-20

- out-of-band, 3-10, 3-17, 3-62, 3-88

- receiving, 3-9, 3-82

- sending, 3-9

- unconnected sockets, 3-10

- urgent, 3-17

datagrams

- associations, 3-30, 3-31, 3-105

- datagram error indication, 1-20

- fragmentation, 3-52, 3-101

- receiving, 3-80, 3-82

- sending, 3-90, 3-106

- use with connectionless sockets, 3-6, 3-9, 3-105

descriptor

- file, 3-32, 3-77, 3-108

- I/O, 3-66, 3-88

- socket, 3-105

- with select() call, 3-66, 3-88

differences, UNIX vs. MVS, 3-14

Directory Services Parameter List. *See* DPL., 5-2

DIRSRV macro, 5-2
dirsrv(), 5-1
disconnect indication, 1-20
DNR, Domain Name Resolver
 definition of, 5-1
 requests to, 5-1
Domain Name Resolver. See DNR., 5-1
dot notation, 3-58
DPL, Directory Services Parameter List, 5-1, 5-2
dpl struct, 5-4
dsects, correspondence to C structs, 2-1

E

ECB
 list, 3-66
 posting, 3-66
endpoint
 closing of, 1-10
 creation of, 1-15, 3-105
 current state of, 1-16
 ID, 1-15
 passing, 1-10, 1-15
 range, 3-17
environment variables
 LSCNCOM, A-6
errno, 3-18, 3-79, 3-109
errno.h, 3-19, 4-1
error
 analysis with `terror()`, 1-11
 checking in C library, 1-3
 code conversion, 3-18
 codes, 3-18, 3-20, 4-3
 logic, 1-23
 messages using `perror()`, 3-79
 TPL-based request, 1-22
`error()`, 1-14
ESCONFIG, 3-20
ESYS, 3-21
ETPEND, 3-21
exception condition, 3-66, 3-88

Exit List Structure, 2-17
exits
 `api_end_exit()`, 1-21
 asynchronous, 1-2
 calling sequence, 1-18
 compiler differences, 1-18
 compilers for, 1-3
 LERAD, 1-9
 `lerad_error_exit()`, 1-23
 `protocol_event_exit()`, 1-20
 SYNAD, 1-9
 `synad_error_exit()`, 1-22
 `tpl_completion_exit()`, 1-19
 `transport_provider_end_exit()`, 1-21
expedited data indication, 1-20

F

file descriptor, 3-32, 3-77, 3-108
FIONBIO, 3-14
fork, 3-15
FREEMAIN macro instruction, 1-14
function
 format, 1-6, 3-23
 prototypes, 1-24, 3-19

G

GET_ERRNO, 3-18
`getpeername()`, 3-8
`getsockopt`, UNIX System Services options, 7-5
`getsockopt()`, 3-13

H

h_errno, 3-37

header files

- installation of, 2-2
- introduction to, 1-2
- socket library, 6-3
- summary, 4-1

host

- address, 3-35
- name, 3-34, 3-37
- UNIX, 3-35
- UNIX System Services resolution, 7-8

hostent struct, 3-35, 4-3

I

I/O

- blocking/non-blocking, 3-14
- controls, 3-62
- descriptor, 3-66, 3-88
- file functions, 3-12, 3-21
- non-blocking, 3-32, 3-62, 6-13
- techniques for integrating with UNIX, 3-22

IF#ARP, 4-1

IF#ETHER, 4-2

if.h, 4-1

if_arp.h, 4-1

if_ether.h, 4-2

in.h, 4-2

in_addr, 3-7

in_addr struct, 4-2

include files, 4-1

INET support, 7-18

inet.h, 3-19, 4-2

installation

- data sets, 6-2
- load library, 6-5
- socket and C libraries, 6-2

internet address, 3-58, 3-60

ioccom.h, 4-2

ioctl(), 3-14, 3-18

ioctl.h, 4-2

iovcnt struct, 3-12

iovec, 4-4

iovec struct, 3-11

ip.h, 3-19, 4-3

J

JCL, 6-6

CLGIBMCM, 6-6, 6-18

CLGIBMCR, 6-6, 6-19

CLGSASC, 6-20

CLGSASCR, 6-6

CLGSASR, 6-21

UMODCFGI, 6-6, 6-14, 6-15

UMODCFG, 6-6, 6-14, 6-17

K

keyword facility of macro instructions, 1-5

L

L\$CNCOM, LSCNCOM alias, A-5

lerad_error_exit(), 1-23

linger struct, 4-3

listen(), 3-8

load library, 6-5

LOADXL8 command, 7-14

log, 3-73, 3-98, 3-109

logic error, 1-23

LSCNCOM

alias

L\$CNCOM, A-5

configuration files, A-7

environment variables, A-6

requirements, A-4

restrictions, A-3

SAS/C socket interface, A-1

usage, A-5

M

macro instructions
 ACLOSE, 1-7
 AOPEN, 1-8
 comparison with C API functions, 1-2, 1-5
 FREEMAIN, 1-14
 keyword, 1-5
 TCHECK, 1-9
 TCLOSE, 1-10
 TERROR, 1-11
 TEXEC, 1-12
 TOPEN, 1-15
 TSTATE, 1-16
 WTO, 1-17

maximum number of sockets, 3-55

mbufs, 6-13

message interface to WTO, 1-17

MSG_OOB, 3-17

msg_hdr struct, 3-11, 4-3

MVS
 ATTACH, 3-15
 front-ending UNIX functions, 3-22
 I/O operations, 3-21
 porting socket programs to, 3-2

mvselect(), 3-18

N

netdb.h, 3-19, 4-3

netent struct, 4-3

network
 byte order, 3-56, 3-57, 3-71, 3-72
 name, 3-38
 number, 3-40

network address, 3-60

non-blocking I/O, 6-13

non-blocking mode, 3-14, 3-66, 3-88

non-reentrant applications, 6-5, 6-7, 6-14, 6-18

normal data indication, 1-20

NOSLIBCK, 3-19

O

object modules
 C library functions included as, 1-3
 installation of, 1-3
 usage of, 6-5

OpenEdition. *See* UNIX System Services

openold(), 3-18, 3-29

options
 IP, 3-20
 socket, 3-13, 3-20, 3-49, 3-99, 3-106, 4-3
 TPL, 1-10
 UDP, 3-20

orderly release, 1-20, 6-13

out-of-band data, 1-20, 3-10, 3-62, 3-88

P

param.h, 4-3

Partitioned Data Sets (PDS), 6-2

passing control to another socket, 3-15, 3-18, 3-29, 3-69

PDS, Partitioned Data Sets, 6-2

peer, 3-42

perror(), 3-109

PFS
 installation, 7-3
 transport driver, 7-3

PFSLOAD data set, 7-3

Physical File System, installation, 7-3

preprocessor, 1-5

priority, 3-98

priority, 3-109

privilege, 3-14

protocol
 address binding, 1-12
 event exit, 1-20
 events, 1-20
 families, 3-108, 4-3
 name, 3-44
 number, 3-43
 options, 3-13
 selection, 3-105
protocol_event_exit(), 1-20
protoent, 4-3
prototype statements, 1-3

R

read(), 3-12
readv(), 3-12
receiving data, 3-9
recv(), 3-10
recvfrom(), 3-10
recvmsg(), 3-11
reentrant applications, 6-5, 6-7, 6-14, 6-18
Referreferenes,UNIX System
Services/OpenEditionences, 7-2

S

s0skcfg, 4-3
SAS
 socket setup, A-7
 workaround, A-5
SAS/C socket interface, A-1
send(), 3-10
sending data, 3-9
sendmsg(), 3-11
sendto(), 3-9
serrno.h, 4-3
servent, 4-3
service, 3-45, 3-47

setlogmask(), 3-98
setsockopt(), 3-13, 6-13, 7-6
shutdown(), 3-13
sin_addr, 3-7
SO_LINGER, 6-13
SOCK_ASSOC, 3-5, 3-16
SOCK_CONNLESS, 3-5, 3-16
SOCK_DGRAM, 3-4, 3-16
SOCK_RAW, 3-5, 3-16
SOCK_RDM, 3-4
SOCK_SEQPACKET, 3-4
SOCK_STREAM, 3-4, 3-16
sockaddr struct, 4-3
sockcf.c, 6-6, 6-7, 6-14
sockcfg struct, 6-22
sockcfg.h, 3-19, 4-3
socket interface, SAS/C, A-1
socket(), 3-6, 6-13
socket.h, 3-3, 3-19, 4-3
sockets
 association, 3-9
 binding, 3-7, 3-17
 blocking/non-blocking, 3-14
 buffering limits, 6-13
 changing values, 6-22
 client mode, 3-9
 config file sockcf, 6-6
 configuration, 3-20, 6-7, 6-14
 connecting, 3-6
 creation, 3-6
 customization, 6-22
 definition of, 3-3
 descriptor, 3-6, 3-8, 3-27, 3-105
 differences between UNIX and MVS, 3-14
 duplication of messages, 3-3
 error codes, 3-18
 error messages, 3-79
 header files, 4-1
 I/O file functions, 3-12, 3-62
 installation, 6-2
 introduction, 3-2
 life cycle, 6-13
 maximum number, 3-55

- name, 3-8, 3-26, 3-48
- non-blocking mode, 3-66, 3-88
- options, 3-13, 3-20, 3-49, 3-99, 3-106, 4-3
- orderly release, 6-13
- passing control, 3-15, 3-18, 3-29, 3-69
- privilege, 3-14
- receiving data, 3-9
- references, 3-3
- sending data, 3-9
- server mode, 3-8
- shutting down connection, 3-13, 3-104
- termination, 3-13, 3-15, 3-104
- types, 3-4, 3-16, 3-105, 4-3
- UNIX, 3-14, 3-108
- user mode, 3-14
- using the socket library, 6-18

sockio.h, 4-3

sockproto struct, 4-3

sockvar.h, 3-20, 4-4

sstat struct, 4-3

stopping and starting sockets, 7-17

synad_error_exit(), 1-22

SYSLIB, 1-2, 6-5

syslog.h, 4-4

system log, 3-109

systemtime.h, 4-4

T

TCHECK macro instruction, 1-9

tcheck(), 1-3, 1-9, 1-19, 1-22

TCLOSE macro instruction, 1-10

tclose(), 1-3, 1-10

TCP, Transmission Control Protocol, 3-5, 3-16, 3-49, 3-53, 3-99, 3-102, 3-105, 6-10, 6-13

tcp.h, 3-20, 4-4

TEM, C struct correspondence, 2-1

tem struct, 2-1, 2-5

termination

- address space, 6-13
- API subsystem, 1-21
- C library function, 1-7

- orderly release, 6-13
- orderly release indication, 1-20
- routine, 3-15
- transport provider, 1-21

TERROR macro instruction, 1-11

terror(), 1-3, 1-11, 1-17

TEXEC macro instruction, 1-12

texec(), 1-3, 1-12

tferror(), 1-4, 1-14

TIB, C struct correspondence, 2-1

tib struct, 2-1, 2-6

time.h, 4-4

TOPEN macro instruction, 1-15

topen(), 1-4, 1-15

TPA, C struct correspondence, 2-1

tpa struct, 2-1, 2-7

TPL

- activation of, 1-9, 1-16
- completion exit, 1-19
- completion status of, 1-9
- errors, 1-22
- execution of, 1-12
- format of, 1-5
- inactivation, 1-19
- logic error, 1-23
- option code field, 1-10
- Transport Service Parameter List, 1-5
- usage, 1-5

TPL C struct correspondence, 2-1

tpl struct, 1-5, 2-1, 2-7

tpl_completion_exit(), 1-19

tplepids, 1-12

TPLEPID, 1-5

tplfnccd, 1-12

tplopced3, 1-10

TPO, C struct correspondence, 2-2

tpo struct, 2-2, 2-14

translate table conversion, LOADXL8 command, 7-13, 7-14

Transmission Control Protocol. See TCP., 3-5

Transport Endpoint Error Message, 2-5
Transport Endpoint State Word, 2-15
Transport Endpoint User Block, 2-16
Transport Exit Parameter List, 2-18
Transport Protocol Address, 2-7
Transport Protocol Options, 2-14
transport provider, 1-15, 1-21
Transport Service Information Block, 2-6
transport service parameter list. *TPL*
Transport Service Parameter List. See *TPL.*, 2-7
transport_provider_end_exit(), 1-21
TSTATE macro instruction, 1-16
tstate(), 1-4, 1-16
TSW, C struct correspondence, 2-2
tsw struct, 2-2, 2-15
TUB, C struct correspondence, 2-2
tub struct, 2-2, 2-16
twto(), 1-4, 1-17
TXL, C struct correspondence, 2-2
txl struct, 2-2, 2-17
TXP, C struct correspondence, 2-2
txp struct, 1-21, 2-2, 2-18
types of sockets, 3-105, 4-3
types.h, 4-4

U

UDP, User Datagram Protocol, 3-16, 3-20, 3-25, 3-105, 6-10
uio struct, 4-4
uio.h, 3-20, 4-4
UMODCFGI, 6-6, 6-14, 6-15

UMODCFGS, 6-6, 6-14, 6-17
unistd.h, 4-4
UNIX
 BSD, 3-2
 buffering limits, 6-13
 front-ending I/O functions, 3-22
 hosts, 3-35
 I/O routines, 3-21
 sockets, 3-14, 3-108
UNIX System Services
 CONVXL8 command, 7-13
 getsockopt options, 7-5
 host name/address resolution, 7-8
 installation, 7-3
 ioctl parameters, 7-4
 limitations, 7-17
 LOADXL8 command, 7-14
 security, 7-3
 setsockopt, 7-6
 startup, 7-3
 stopping and starting sockets, 7-17
 translate table conversion, 7-13, 7-14
urgent data, 3-17
USEAPIREQBYTECNTDEFS, 6-14
user mode, 3-14
user.h, 3-20
USESASCTIME, 3-55

V

va_arg, 3-110

W

write(), 3-12
writev(), 3-12
WTO macro instruction, 1-17



Computer Associates™