SOCKETS API Reference

SOCKETS API Overview

SOCKETS API is compatible with a wide range of third party TCP/IP applications, and contains descriptions for each of the supported functions. The function descriptions are preceded by introductory information that provides some background on the implementation of the SOCKETS API. The definitions and prototypes for the C environment are supplied in CAPI.H and COMPILER.H, while the implementation of the C interface in CAPI.C and _CAPI.C. The SOCKETS API provides an interface to the socket, name resolution, ICMP ping, and kernel facilities provided by the Datalight DOS SOCKETS product. A *socket* is an end-point for a connection and is defined by the combination of a host address (also known as an IP address), a port number (or communicating process ID), and a transport protocol, such as UDP or TCP. Two connected SOCKETS using the same transport protocol define a connection. The API uses a socket handle, sometimes referred to as simply a socket. Previously, the socket handle has been referred to as a network descriptor. The socket handle is required by most function calls in order to access a connection. Two types of SOCKETS can be used: 1) a DOS compatible socket, previously referred to as a local network descriptor, which uses a DOS file handle, and 2) a normal socket (previously referred to as a global network descriptor) which does not use a DOS file handle. New designs should always use normal SOCKETS. A socket handle is obtained by calling the **GetSocke**t() function. A socket handle can only be used for a single connection. When no longer required, such as when a connection has been closed, the socket handle must be released by calling **ReleaseSocke**t(). DOS compatible socket handles are in the range 0 to 31, although 0 to 4 are normally be used by the C runtime for DOS files like **stdin** and **stdou**t. Normal socket handles are positive numbers greater than 63.

Types of Service

SOCKETS can be used with one of two service types:

- STREAM (using TCP).
- DATAGRAM (using UDP).

A stream connection provides for the bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries. No broadcast facilities can be used with a stream connection. A datagram connection supports bi-directional flow of data that is not guaranteed to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram connection is that record boundaries in data are preserved. Datagram connections closely model the facilities found in many contemporary packet switched networks such as Ethernet. Broadcast messages may be sent and received.

Establishing Remote Connections

To establish a connection, one side (the server) must execute a **ListenSocket()** and the other side (the client) a **ConnectSocket()**. A connection consists of the local socket / remote socket pair. It is therefore possible to have a connection within a single host as long as the local and remote *port* values differ. Each host in an IP network must have at least one host address also known as an IP address. When a host has more than one physical connection to an IP network, it may have more than one IP address. An IP address must be unique within a network. An IP address is 32 bits in length, a port number 16 bits. A value of zero means “any” while a binary value of all 1s means “all.” The latter value is used for broadcasting purposes. Using the NET_ADDR structure conveys the addresses (host/port) to be used in a connection. The local host is not specified; it is implied. If a value of 0 is specified for *dwRemoteHos*t, any remote IP address is accepted; and if a value of 0 is specified for a remote port, any remote port is accepted. This is normally the case when a server is listening for an incoming call. If a value of 0 is specified for *wLocalPort* in the case of a client calling **ConnectSocket(**), a unique port number is assigned by the TCP/IP stack.

Using STREAM and DATAGRAM Services

When using the STREAM service (TCP), bi-directional data can be sent using the **WriteSocket()** function and received using the **ReadSocket()** function until one side performs an **EofSocket()** after which that side cannot send any more data , but can still receive data until the other side performs an **EofSocket**(), **AbortSocket()** or **ReleaseSocket**().When using the DATAGRAM service, datagrams can be sent without first establishing a "connection". In fact UDP provides a "connectionless" service although the connection paradigm is used. In addition to **ReadSocket()** and **WriteSocket**(), **ReadFromSocket()** and **WriteToSocket()** can be used. In this case **EofSocket()** has no meaning and returns an error.

Blocking and Non-blocking Operations

The default behavior of socket functions is to block on an operation and only return when the operation has completed. For example, the **ConnectSocket()** function only returns after the connection has been performed or an error is encountered. This behavior applies to most socket function calls, such as **ReadSocket()** and even **WriteSocket**(), and especially on STREAM connections. In many, if not most applications, this behavior is unacceptable in the single-threaded DOS environment and must be modified. This modification can be accomplished in by either::

1. By specifying the NET_FLG_NON_BLOCKING flag on **ReadSocket()** and **WriteSocket()** calls or
2. By making all operations on a socket non-blocking by calling **SetSocketOption()** with the NET_OPT_NON_BLOCKING option.

If a non-blocking operation is performed, the function always returns immediately. If the function could not complete without blocking, an error is returned with *SocketsErrNo* containing ERR_WOULD_BLOCK. This error should be regarded as a recoverable error and the operation should be retried, preferably at some later time.

Blocking Operations with Timeouts

A possible alternative to using non-blocking operations is to use blocking operations with timeouts. This is done by calling **SetSocketOption()** with the NET_OPT_TIMEOUT option, in which case the function blocks for the specified time, or until completed, whichever occurs first. If the specified timeout occurs first, an error is returned with *SocketsErrNo* containing ERR_TIMEOUT and the operation must be retried. Use non-blocking operations rather than timeouts, although they may be somewhat more difficult to implement.

Asynchronous Notifications/Callbacks

Asynchronous notifications or callbacks can be used in cases where the polling implied by non-blocking operation is not desirable, when immediate action is required, when a network operation completes, or when a SOCKETS application runs as a TSR. However, such notifications may be difficult to use and the programmer must be careful to avoid system crashes resulting from improper use. The **SetAsyncNotification()** function sets functions to be called on specific events, such as opening and closing of STREAM connections and receiving data on STREAM and DATAGRAM connections. The **SetAlarm()** function is called to set a function to be called when a timer expires. Asynchronous notifications are disabled by the **DisableAsyncNotification()** function and enabled by the **EnableAsyncNotification()** function. For more details on the operation and pitfalls associated with callbacks, refer to the description of **SetAsyncNotification**(). **ResolveName**(), **GetDCSocket**(), **ConvertDCSocket**(), **ReleaseSocket()** on a DC socket, **ConnectSocket()** with a socket value of –1, and **ListenSocket()** with a socket value of –1 all call DOS. For this reason, these functions should not be called from within a callback or an interrupt service routine.

IP Address Resolution

Two functions are provided for IP address resolution. **ParseAddress()** converts a dotted decimal address to a 32-bit IP address.

ResolveName() converts a symbolic host name to a 32-bit IP address using a host table lookup; if that fails and a domain server is configured, then to a DNS lookup. **ResolveName()** calls DOS to perform a host table lookup and always blocks while doing a DNS lookup.

Obtaining SOCKETS Kernel Information

You can obtain information on the SOCKETS TCP/IP kernel by the **GetKernelInformation()**, **GetVersion()** and **GetKernelConfig()** functions. You can unload the kernel by **ShutDownNet()**.

Error Reporting

In general, the C functions implementing the compatible API return a value of –1 if the return type is **int** and an error is encountered, in which case, the actual error code is returned in a common variable *SocketsErrNo*. In some cases, *iSubNetErrNo* is also used. Any API call may fail with an error code of ERR_API_NOT_LOADED or ERR_RE_ENTRY. ERR_RE_ENTRY is returned when the SOCKETS kernel has been interrupted. This condition can occur only when the API is called from an interrupt service routine. Programs designed for this type of operation, such as TSR programs activated by a real time clock interrupt, should be coded to handle this error by re-trying the function at a later stage.

Low Level Interface to Compatible API

Low level functions to access the Compatible API may be used. In this case, the compatible API is called by setting up the CPU registers and executing a software interrupt. The default interrupt is 61 hexadecimals, but may be relocated when SOCKETS is loaded. If the actual interrupt is not known, a search may be performed for it. Refer to the source file CAPI.C for more details. On entry, AH contains a number specifying the function to perform. On return, the carry flag is cleared on success and set on failure.

Alternatives to the Compatible API

Two additional programming interfaces are available for use with SOCKETS. The first is an earlier revision of CAPI, now called CAPIOLD. This interface is provided to maintain compatibility with applications developed for SOCKETS 1.0. It is superceded by CAPI, which is better-documented and easier to use. Both CAPI and CAPIOLD rely on an internal array of socket descriptors, which must be configured at compile-time. This can use excess memory if your application rarely uses a large number of SOCKETS simultaneously. In addition, it is advised that these APIs do not deal well with mixing both blocking and non-blocking SOCKETS in one application. The second interface is called APIC, and is more robust for TSR and server applications. It is a more natural stack interface, which hides fewer details from the programmer. As a result, it is more difficult to work with, and should be used only when its extended features and lowered memory footprint are required. Due to the low popularity of this interface, documentation is provided only inside the APIC.C and APIC.H source files.

Porting for Compilers

Compiler specific functions have been written into the compiler.h. Modifications for compilers other than the supplied Borland BC5.2 compiler and any listed within compiler.h need to happen within this file. Datalight will offer any assistance we can to help with porting to other compilers but our expertise exists within the supplied Borland compiler.

Usage Notes

Please refer to the make file provided within the SOCKETS\EXAMPLES directory for command line compiler options.

Function Reference

The following sections describe the individual functions of the SOCKETS API.

DisableAsyncNotification

The **DisableAsyncNotification()** function disables Asynchronous notifications (callbacks).

C syntax

int DisableAsyncNotification(void);

Return value

Returns –1 on error with *SocketsErrNo* containing the error. Returns previous state on success, 0 for disabled, 1 for enabled.

Low level calling parameter

AH DISABLE_ASYNC_NOTIFICATION (0x11)

Low level return parameter

AX = previous state, 0 = disabled, 1 = enabled

EnableAsyncNotification

The **EnableAsyncNotification()** function enables asynchronous notifications (callbacks).

C syntax

int EnableAsyncNotification(void);

Return value

Returns –1 on error with SocketsErrNo containing the error. Returns previous state on success, 0 for disabled, 1 for enabled.

Low level calling parameter

AH ENABLE_ASYNC_NOTIFICATION (0x12)

Low level return parameter

AX = previous state, 0 = disabled, 1 = enabled.

GetAddress

The **GetAddress()** function gets the local IP address of a connection. In the case of a single interface host, this is the IP address of the host. In the case of more than one interface, the IP address of the interface being used to route the traffic for the specific connection is given.

C syntax

DWORD GetAddress(int iSocket);

Parameter

iSocket

Socket handle for the connection.

Return value

Returns IP address on success. Returns 0L on error with *SocketsErrNo* containing the error.

Low level calling parameters

AH GET_ADDRESS (0x05)

BX Socket

Low level return parameters

AX:DX = IP address of this host. AX:DX = 0:0 on error.

GetPeerAddress

The **GetPeerAddress()** function gets peer address information on a connected socket.

C syntax

int **GetPeerAddress(**int _iSocket_, NET_ADDR *_pAddr_);

Options

iSocket

Socket handle for the connection.

pAddr

Pointer to NET_ADDR structure to receive information.

Return values

Returns 0 and NET_ADDR structure filled in on success. Returns –1 with *SocketsErrNo* containing the error on failure.

Low level calling parameters

AH GET_PEER_ADDRESS (0x16).

BX Socket.

DS:DX Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, the address structure is filled in

If carry flag is set, AX = error code

GetKernelInformation

The **GetKernelInformation()** function gets specified information from the kernel.

C syntax

int **GetKernelInformation(**int _iReserved_,BYTE _bCode_,BYTE _bDevID_,void *_pData_,WORD

*_pwSize_);

Options

iReserved

Reserved value, set to zero.

bCode

Code specifying kernel info to retrieve:

K_INF_HOST_TABLE Gets name of file containing host table.

K_INF_DNS_SERVERS Gets IP addresses of DNS Servers.

K_INF_TCP_CONS Gets number of Sockets (DC + normal).

K_INF_BCAST_ADDR Gets broadcast IP address.

K_INF_IP_ADDR Gets IP address of first interface.

K_INF_SUBNET_MASK Gets netmask of first interface.

pData

Pointer to data area to receive kernel information.

puSize

Pointer to WORD containing length of data area.

Return values

On success returns 0 with data area and size word filled in. Returns –1 with *SocketsErrNo* containing the error on failure.

Low level calling parameters

AH GET_KERNEL_INFO (0x02)

DS:SI Pointer to data area to receive kernel information.

ES:DI Pointer to WORD containing length of data area.

DH Code specifying kernel info to retrieve.

K_INF_HOST_TABLE Gets name of file containing host table.

K_INF_DNS_SERVERS Gets IP addresses of DNS Servers.

K_INF_TCP_CONS Gets number of Sockets (DC + normal).

K_INF_BCAST_ADDR Gets broadcast IP address.

K_INF_IP_ADDR Gets IP address of first interface.

K_INF_SUBNET_MASK Gets netmask of first interface.

Low level return parameters

If no error, data area is filled in as well as the size word.

GetVersion

The **GetVersion()** function gets version number of the Compatible API.

C syntax

int GetVersion(void);

Return value

Returns 0x214 on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH GET_NET_VERSION (0x0F).

Low level return parameters

AX = 0x214

ICMPPing

The **ICMPPing()** function sends an ICMP ping (echo request) and waits until a response is received or for six seconds if no response is received. **ICMPPing()** is always a blocking function.

C syntax

int **ICMPPing(**DWORD *dwHos*t, int *iLengt***h**);

Options

dwHost

IP address of host to ping.

iLength

Number of data bytes in ping request.

Return value

Returns 0 on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH ICMP_PING (0x30).

CX number of data bytes in ping request.

DX:BX IP address of host to ping.

Low level return parameters

If carry flag is set, AX = error code.

IsSocket

The **IsSocket()** function checks a DOS compatible socket for validity.

C syntax

int **IsSocket(**int *iSocke*t);

Parameter

iSocket

DOS Compatible socket handle for the connection.

Return value

Returns 0 on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH IS_SOCKET (0x0D).

BX Local socket.

Low level return parameters

Carry flag clear if valid.

Carry flag set if not valid, AX = error code.

GetDCSocket

The **GetDCSocket()** function gets a DOS-compatible socket handle. This function calls DOS to open a DOS file handle.

C syntax

int GetDCSocket(void);

Return value

Returns socket handle on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH GET_DC_SOCKET (0x22).

Low level return parameters

If carry flag is clear, AX = Socket.

If carry flag is set, AX = error code.

GetSocket

The **GetSocket()** function gets a socket handle.

C syntax

int GetSocket(void);

Return value

Returns socket handle on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH GET_SOCKET (0x29).

Low level return parameters

If carry flag is clear, AX = Socket.

If carry flag is set, AX = error code.

GetKernelConfig

The **GetKernelConfig()** function gets the kernel configuration.

C syntax

int **GetKernelConfig(**KERNEL_CONFIG _*psKc_);

Parameter

psKc

Pointer to KERNEL_CONFIG structure.

bKMaxTcp Number of TCP sockets allowed.

BKMaxUdp Number of UDP sockets allowed.

bKMaxIp Number of IP sockets allowed (0).

bKMaxRaw Number of RAW_NET sockets allowed (0).

bKActTcp Number of TCP sockets in use.

bKActUdp Number of UDP sockets in use.

bKActIp Number of IP sockets in use (0).

bKActRaw Number of RAW_NET sockets in use (0).

wKActDCS Number of active Dos Compatible Sockets.

wKActSoc Number of active normal Sockets.

bKMaxLnh Maximum header on an attached network.

bKMaxLnt Maximum trailer on an attached network.

bKLBUF_SIZE Size of a large packet buffer.

bKNnet Number of network interfaces attached.

dwKCticks Milliseconds since kernel started.

dwKBroadcast IP broadcast address in use.

Return value

Returns 0 on success with KERNEL_CONFIG structure filled in, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH GET_KERNEL_CONFIG (0x2A).

DS:SI pointer to kernel_conf structure.

Return

KERNEL_CONF structure filled in.

ConvertDCSocket

The **ConvertDCSocket()** function changes a DOS compatible socket handle into a normal socket handle. This function calls DOS to close a DOS file handle.

C syntax

int ConvertDCSocket(int *iSocket*);

Parameter

iSocket

DOS Compatible socket handle.

Return value

Returns socket handle on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH CONVERT_DC_SOCKET (0x07).

BX Local socket.

Low level return parameters

AX = Global socket if no error.

GetNetInfo

The **GetNetInfo()** function gets information about the network.

C syntax

int **GetNetInfo(**int *iSocket*, NET_INFO *psNI);

Parameter

iSocket

Socket handle for the connection.

psNI

Pointer to NET_INFO structure. The following members of NET_INFO are obtained:

DwIPAddress

dwIPSubnet

iUp

iLanLen

pLanAddr

Return value

Returns 0 with NET_INFO structure filled in on success, -1 on failure with *SocketsErrNo*

containing the error code.

Low level calling parameters

AH GET_NET_INFO (0x06).

DS:SI Pointer to netinfo structure.

Low-level return

netinfo structure filled in.

ConnectSocket

The **ConnectSocket()** function makes a network connection. If *iSocket* is specified as –1, a DOS compatible socket is assigned. In this case only, DOS is called to open a file handle. If *iSocket* specifies a non-blocking socket or *iType* specifies a DATAGRAM connection, this call returns immediately. In the case of a STREAM connection, the connection may not yet be established. **ReadSocket()** can be used to test for connection establishment. As long as **ReadSocket()** returns an ERR_NOT_ESTAB code, the connection is not established. A good return or an error return with ERR_WOULD_BLOCK indicates an established connection. A more complex method uses **SetAsyncNotify()** with NET_AS_OPEN to test for connection establishment. NET_AS_ERROR should also be set to be notified of a failed open attempt.

C syntax

int **ConnectSocket(**int *iSocket*, int *iType*, NET_ADDR *psAddr);

Parameter

iSocket

Socket handle for the connection.

iType

Type of connection: STREAM or DATAGRAM.

psAddr

Pointer to NET_ADDR structure.

Return value

Returns socket on success, -1 on failure with SocketsErrNo containing the error code.

Low level calling parameters

AH CONNECT_SOCKET (0x13).

BX Socket.

DX Connection mode: Stream or DataGram.

DS:SI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag clear, initiated OK, AX = Socket.

If carry flag set, AX = error code.

ListenSocket

The **ListenSocket()** function listens for a network connection. If *iSocket* is specified as –1, a DOS compatible socket is assigned. In this case only, DOS is called to open a file handle. If *iSocket* specifies a non-blocking socket or *iType* specifies a DATAGRAM connection, this call returns immediately. In the case of a STREAM connection, the connection may not be established yet. **ReadSocket()** can be used to test for connection establishment. As long as **ReadSocket()** returns an ERR_NOT_ESTAB code, the connection is not established. A good return or an error return with ERR_WOULD_BLOCK indicates connection establishment. A more complex method is to use **SetAsyncNotify()** with NET_AS_OPEN to test for connection establishment. NET_AS_ERROR should also be set to be notified of a failed open attempt.

C syntax

int **ListenSocket(**int *iSocke*t, int *iTyp*e, NET_ADDR _*psAdd_**r**);

Parameter

iSocket

Socket handle for the connection.

iType

Type of connection: STREAM or DataGram.

psAddr

Pointer to NET_ADDR structure.

Return value

Returns socket handle on success, -1 on failure with SocketsErrNo containing the error code.

Low level calling parameters

AH LISTEN_SOCKET (0x23).

BX Socket.

DX Connection mode: STREAM or DataGram.

DS:SI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag clear, initiated OK, AX = Socket.

If carry flag set, AX = error code.

SelectSocket

The **SelectSocket()** function tests all DOS compatible sockets for data availability and readiness to write. A 32-bit DWORD representing 32 DC sockets is filled in for each socket with receive data, and another 32-bit DWORD for DC sockets ready for writing. The least-significant bit represents the socket with value 0 and the most-significant bit represents the socket with value 31. Bits representing unused sockets are left unchanged.

C syntax

int **SelectSocket(**int *iMaxi*d, long **plIflag*s, long **plOflag***s**);

Options

iMaxid

Number of sockets to test.

plIflags

Pointer to input flags indicating receive data availability.

plOflags

Pointer to output flags indicating readiness to write.

Return value

Returns 0 on success with *plIflags and *plOflags filled in with current status, -1 on failure

with SocketsErrNo containing the error code.

Low level calling parameters

AH SELECT_SOCKET (0x0e).

BX Number of sockets to test.

DS:DX Pointer to DWORD for data availability.

ES:DI Pointer to DWORD for readiness to write.

Low level return parameters

Both DWORDs updated with current status.

ReadSocket

The **ReadSocket()** function reads from the network using a socket. **ReadSocket()** returns as soon as any non-zero amount of data is available, regardless of the blocking state. If the operation is non-blocking, either by having used **SetSocketOption()** with the NET_OPT_NON_BLOCKING option or specifying *wFlags* with NET_FLG_NON_BLOCKING, **ReadSocket()** returns immediately with the count of available data or an error of ERR_WOULD_BLOCK. With a STREAM (TCP) socket, record boundaries do not exist and any amount of data can be read at any time regardless of the way it was sent by the peer. No data is truncated or lost even if more data than the buffer size is available. What is not returned on one call, is returned on subsequent calls. If a NULL buffer is specified, the number of bytes on the receive queue is returned. In the case of a DataGram (UDP) socket, the entire datagram is returned in one call, unless the buffer is too small in which case the data is truncated, thereby preserving record boundaries. Truncated data is lost. If data is available and both the NET_FLG_PEEK and NET_FLG_NON_BLOCKING flags are specified, the number of datagrams on the receive queue is returned. If data is available and NET_FLG_PEEK is set and a NULL buffer is specified, the number of bytes in the next datagram is returned.

C syntax

int **ReadSocket(**int *iSocket*, char **pcBuf*, WORD *wLen*, NET_ADDR **psFrom*, WORD

*wFlag***s**);

Options

iSocket

Socket handle for the connection.

pcBuf

Pointer to buffer to receive data.

wLen

Length of buffer, i.e. maximum number of bytes to read.

psFrom

Pointer to NET_ADDR structure to receive address information about local and remote ports and remote IP address.

wFlags

Flags governing operation. Any combination of:

NET_FLG_PEEK Don't dequeue data.

NET_FLG_NON_BLOCKING Don't block.

Return value

Returns number of bytes read on success, -1 on failure with *SocketsErrNo* containing the error code. A return code of 0 indicates that the peer has closed the connection.

Note: If blocking is disabled, a failure with an error code of ERR_WOULD_BLOCK is completely normal and only means that no data is currently available.

Low level calling parameters

AH READ_SOCKET (0x1b).

BX Socket.

CX Maximum number of bytes to read.

DX Flags - any combination of.

NET_FLG_PEEK: Don't dequeue data.

NET_FLG_NON_BLOCKING: Don't block.

DS:SI Pointer to buffer to read into.

ES:DI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, AX = CX = number of bytes read.

If carry flag is set, AX = error code.

ReadFromSocket

The **ReadFromSocket()** function reads from the network using a socket and is only intended to be used on DataGram sockets. All datagrams from the IP address and port matching the values in the NET_ADDR structure are returned while others are discarded. A zero value for *dwRemoteHost* is used as a wildcard to receive from any host and a zero value for *wRemotePort* is used as a wildcard to receive from any port. The local port, *wLocalPort* , can not be specified as zero. In other respects **ReadFromSocket()** behaves the same as **ReadSocket**().

C syntax

int **ReadFromSocket(**int *iSocke*t, char **pcBu*f, WORD *wLe*n, NET_ADDR **psFro*m, WORD *wFlag*s);

Options

iSocket

Socket for the connection.

pcBuf

Pointer to buffer to receive data.

wLen

Length of buffer, i.e. maximum number of bytes to read.

psFrom

Pointer to NET_ADDR structure to receive address information about local and remote ports and remote IP address.

wFlags

Flags governing operation. Any combination of:

NET_FLG_PEEK Don't dequeue data.

NET_FLG_NON_BLOCKING Don't block.

Return value

Returns number of bytes read on success, -1 on failure with *SocketsErrNo* containing the error code. A return code of 0 indicates that the peer has closed the connection. Note the following anomaly: If blocking is disabled, a failure with an error code of ErrWouldBlock is completely normal and only means that no data is currently available.

Low level calling parameters

AH READ_FROM_SOCKET (0x1d).

BX Socket.

CX Maximum number of bytes to read.

DX Flags - any combination of.

NET_FLG_PEEK:- Don't dequeue data.

NET_FLG_NON_BLOCKING:- Don't block.

DS:SI Pointer to buffer to read into.

ES:DI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, AX = CX = number of bytes read.

If carry flag is set, AX = error code.

WriteSocket

The **WriteSocket()** function writes to the network using a socket.

C syntax

int **WriteSocket(**int *iSocke*t, char **pcBu*f, WORD *wLe*n, WORD *wFlag*s);

Parameter

iSocket

Socket handle for the connection.

pcBuf

Pointer to buffer to containing send data.

wLen

Length of buffer, i.e. number of bytes to write.

wFlags

Flags governing operation; can be any combination of:

NET_FLG_OOB Send out of band data (TCP only).

NET_FLG_PUSH Disregard Nagle heuristic (TCP only).

NET_FLG_NON_BLOCKING Don't block.

NET_FLG_BROADCAST Broadcast data (UDP only).

NET_FLG_MC_NOECHO Suppress the local echo of a multicast datagram.

Return value

Returns number of bytes written on success, -1 on failure with *SocketsErrNo* containing the error code. The number of bytes actually written on a non-blocking write, can be less than wLen. In such a case, the writing of the unwritten bytes must be retried, preferably after some delay.

Low level calling parameters

AH WRITE_SOCKET (0x1a).

BX Socket.

CX Byte count.

DX Flags - any combination of the following:

NET_FLG_OOB:- Send out of band data (TCP only).

NET_FLG_PUSH:- Disregard Nagle heuristic (TCP only).

NET_FLG_NON_BLOCKING:- Don't block.

NET_FLG_BROADCAST:- Broadcast data (UDP only).

DS:SI Pointer to buffer to write.

Low level return parameters

If carry flag is clear, AX = number of bytes sent.

If carry flag is set, AX = error code.

WriteToSocket

The **WriteToSocket()** function writes to the network using a network address (UDP only).

C syntax

int **WriteToSocket(**int *iSocket*, char _*pcBuf_, WORD *wLen*, NET_ADDR _*psTo_, WORD *wFlags*);

Options

iSocket

Socket handle for the connection.

pcBuf

Pointer to buffer containing send data.

wLen

Length of buffer, i.e. number of bytes to write.

psTo

Pointer to NET_ADDR structure containing local port to write from and remote port and IP address to write to.

wFlags

Flags governing operation. Any combination of:

NET_FLG_NON_BLOCKING Don't block.

NET_FLG_BROADCAST Broadcast data (UDP only).

Return value

Returns number of bytes written on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH WRITE_TO_SOCKET (0x1C).

BX Socket.

CX Byte count.

DX Flags:

NET_FLG_NON_BLOCKING:- Don't block

NET_FLG_BROADCAST:- Broadcast data (UDP only).

DS:SI Pointer to buffer to write.

ES:DI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, AX = number of bytes sent.

If carry flag is set, AX = error code.

EofSocket

The **EofSocket()** function closes the STREAM (TCP) connection (sends a FIN). After **EofSocket()** has been called, no **WriteSocket()** calls may be made. The socket remains open for reading until the peer closes the connection.

C syntax

int **EofSocket(**int *iSocket*);

Parameter

iSocket

Socket handle for the connection.

Return value

Returns 0 on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH EOF_SOCKET (0x18)

BX Socket

Low level return parameters

If carry flag is set, AX = error code.

FlushSocket

The **FlushSocket()** function flushes any output data still queued for a TCP connection.

C syntax

int **FlushSocket(**int *iSocke***t**);

Parameter

iSocket

Socket handle for the connection.

Return value

Returns 0 on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH FLUSH_SOCKET (0x1e)

BX Socket

Low level return parameters

If carry flag set, AX = error code.

ReleaseSocket

The **ReleaseSocket()** function closes the connection and releases all resources. On a STREAM (TCP) connection, this function should only be called after the connection has been closed from both sides otherwise a reset (ungraceful close) can result.

C syntax

int **ReleaseSocket(**int *iSocke***t**);

Parameter

iSocket

Socket handle for the connection.

Return value

Returns socket handle on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH RELEASE_SOCKET (0x08).

BX Socket.

Low level return parameters

AX = error code if carry flag is set

ReleaseDCSockets

The **ReleaseDCSockets** function closes all connections and releases all resources associated with DOS compatible sockets.

C syntax

int ReleaseDCSockets(void);

Return value

Returns 0 on success, -1 on failure with *SocketsErrNo* containing the error code.

C syntax

int ReleaseDCSockets(void);

Low level calling parameters

AH RELEASE_DC_SOCKETS (0x09).

Low level return parameters

AX = error code if carry flag is set.

AbortSocket

The **AbortSocket()** function aborts the network connection and releases all resources. This function causes an unpredictable close (reset) on a STREAM connection.

C syntax

int **AbortSocket(**int *iSocke***t**);

Parameter

iSocket

Socket handle for the connection.

Return value

Returns socket handle on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH ABORT_SOCKET (0x19).

BX Socket.

Low level return parameters

If carry flag is set, AX = error code.

AbortDCSockets

The **AbortDCSockets** function aborts all DOS compatible socket connections.

C syntax

int AbortDCSockets(void);

Return value

Returns 0 on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH ABORT_DC_SOCKETS (0x24).

Low level return parameters

If carry flag is set, AX = error code.

ShutDownNet

The **ShutDownNet()** function shuts down the network and unloads the SOCKETS TCP/IP kernel.

C syntax

int ShutDownNet(void);

Return value

Returns 0 on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH SHUT_DOWN_NET (0x10).

Low level return parameters

None.

SetAlarm

The **SetAlarm()** function sets an alarm timer.

C syntax

int **SetAlarm(**int *iSocke*t, DWORD *dwTim*e, int (far **lpHandle*r)(), DWORD *dwHin***t**);

Options

iSocket

Socket handle for the connection.

dwTime

Timer delay in milliseconds.

lpHandler

Far address of alarm callback. See the description of **SetAsyncNotificatio**n() for the format of the callback function.

dwHint

Argument to be passed to callback function.

Return value

Returns socket handle on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH SET_ALARM (0x2bB.

BX Socket.

CX:DX Timer delay in milliseconds.

DS:SI Address of alarm callback.

ES:DI Argument to be passed to callback.

Low level return parameters

If carry flag is set, AX = error code

See the description of SET_ASYNC_NOTIFICATION for the callback function.

SetAsyncNotification

The **SetAsyncNotification()** function sets an asynchronous notification (callback) for a specific event.

C syntax

int far ***SetAsyncNotification(**int *iSocke*t, int *iEven*t, int (far **lpHandle*r)(),DWORD *dwHin***t**);

Parameter

iSocket

Socket handle for the connection.

iEvent

Event which is being set:

NET_AS_OPEN Connection has opened.

NET_AS_RCV Data has been received.

NET_AS_XMT Ready to transmit.

NET_AS_FCLOSE Peer has closed connection.

NET_AS_CLOSE Connection has been closed.

NET_AS_ERROR Connection has been reset.

lpHandler

Far address of callback function.

dwHint

Argument to be passed to callback function

The handler is not compatible with C calling conventions but is called by a far call with the following parameters:

BX = Socket handle.

CX = Event.

ES:DI = dwHint argument passed to SetAsyncNotification() or SetAlarm().

DS:DX = SI:DX = variable argument depending on event:

NET_AS_OPEN

NET_AS_CLOSE Pointer to NET_ADDR address structure.

NET_AS_FCLOSE

NET_AS_RCV

NET_AS_ALARM Zero.

NET_AS_XMT Byte count which can be sent without blocking.

NET_AS_ERROR Error code –ERR_TERMINATING, ERR_TIME_OUT or

ERR_RESET.

Other CAPI functions may be called in the callback, with the exception of ResolveName() which may call DOS. The callback is not compatible with C argument-passing conventions and some care must be taken. Some CPU register manipulation is required. This can be done by referencing CPU registers, such as _BX, or by means of assembler instructions. In the callback, the stack is supplied by SOCKETS and may be quite small depending on the /s= command line option when loading SOCKETS. The stack segment is obviously not equal to the data segment, which can cause problems when the Tiny, Small or Medium memory model is used. The simplest way to overcome the problem is to use the Compact, Large or Huge memory model. Other options - use the DS != SS compiler option or do a stack switch to a data segment stack . If the callback is written in C or C++, the _loads modifier can be used to set the data segment to that of the module, which destroys the DS used for the variable argument. (This is why DS == SI on entry for SOCKETS version 1.04 and later.) An alternate method is to use the argument passed to SetAsyncNotification() in ES:DI as a pointer to a structure that is accessible from both the main code and the callback. If DS is not set to the data segment of the module, then the functions in CAPI.C do not work: Don't use them in the callback. The callback will probably be performed at interrupt time with no guarantee of reentry to DOS. Do not use any function, such as putchar() or printf(), in the callback which may cause DOS to be called. It is good programming practice to do as little as possible in the callback. The setting of event flags that trigger an operation at a more stable time is recommended. Callback functions do not nest. The callback function is not called while a callback is still in progress, even if other CAPI functions are called. To alleviate the problems in items 2, 3 and 4 above, a handler is provided in CAPI.C that uses the dwHint parameter to pass the address of a C-compatible handler, with a stack that is also C-compatible. This handler is named AsyncNotificationHandler. A user handler named MyHandler below, is called in the normal way with a stack of 500 bytes long. Changing the HANDLER_STACK_SIZE constant in CAPI.C can set the stack size value.

int far **MyHandler(**int *iSocke*t, int *iEven*t, DWORD *dwAr***g**);

SetAsyncNotification(iSocket, iEvent, AsyncNotificationHandler, (DWORD)MyHandle**r**);

Return value

Returns pointer to the previous callback handler on success, -1 on failure with *SocketsErrNo*

containing the error code.

Low level calling parameters

AH SET_ASYNC_NOTIFICATION (0x1F).

BX Socket.

CX Event:

NET_AS_OPEN: Connection has opened.

NET_AS_RCV: Data has been received.

NET_AS_XMT: Ready to transmit.

NET_AS_FCLOSE: Peer has closed connection.

NET_AS_CLOSE: Connection has been closed.

NET_AS_ERROR: Connection has been reset.

DS:DX Address of handler.

ES:DI Argument passed to handler.

Low level return parameters

If carry flag is set, AX = error code, else address of previous handler is returned in DX:AX.

ResolveName

The **ResolveName()** function resolves IP address from symbolic name.

C syntax

DWORD **ResolveName(**char _*pszName_, char _*pcCname_, int _iCnameLen_);

Options

pszName

Pointer to string containing symbolic name.

pcCname

Pointer to buffer to receive canonical name.

ICnameLen

Length of buffer pointed to by *pcName.*

Return value

Returns IP address on success, 0 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH RESOLVE_NAME (0x54).

CX Size of buffer to receive canonical name.

DS:DX Pointer to string containing symbolic name.

ES:DI Pointer to buffer to receive canonical name.

Low level return parameters

If carry flag is clear, AX:DX = IP address.

If carry flag is set, AX = error code.

ParseAddress

The **ParseAddress()** function gets an IP address from dotted decimal addresses.

C syntax

DWORD **ParseAddress(**char _*pszName_);

Parameter

pszName

Pointer to string containing dotted decimal address.

Return value

Returns IP address on success, 0 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH PARSE_ADDRESS (0x50).

DS:DX Pointer to dotted decimal string.

Low level return parameters

AX:DX = IP address.

SetSocketOption

The **SetSocketOption()** function sets an option on the socket.

C syntax

int **SetSocketOption(**int *iSocket*, int *iLevel*, int *iOption*, DWORD *dwOptionValue*, int *iLen***)**;

Options

iSocket

Socket handle for the connection.

iLevel

Level of option. This value is ignored.

iOption

Option to set.

NET_OPT_NON_BLOCKING Set blocking off if *dwOptionValue* is non-zero.

NET_OPT_TIMEOUT Set the timeout to *dwOptionValue* milliseconds. Turn off timeout if

dwOptionValue is zero.

NET_OPT_WAIT_FLUSH Wait for flush if *dwOptionValue* is non-zero.

dwOptionValue

Option value.

iLen

Length of *dwOptionValue*, 4 in all cases.

Return value

Returns global socket on success, -1 on failure with *SocketsErrNo* containing the error code.

Low level calling parameters

AH SET_OPTION (0x20).

BX Socket.

DS:DX Value of option.

DI Option:

NET_OPT_NON_BLOCKING:- Set blocking off if *dwOptionValue* is non-zero.

NET_OPT_TIMEOUT:- Set the timeout to dwOptionValue milliseconds. Turn off timeout if

dwOptionValue is zero.

NET_OPT_WAIT_FLUSH:Wait for flush if dwOptionValue is non-zero.

Low level return parameters

If carry flag is set, AX = error code.

JoinGroup

The **JoinGroup()** function causes SOCKETS to join a multicast group.

C syntax

int **JoinGroup(**DWORD *dwGroupAddress*, DWORD *dwIPAddress*);

Options

dwGroupAddress

The group address on which to receive multicast datagrams.

dwIPAddress

The IP address for the interface to use. The first interface to be specified in SOCKET.CFG

is the default interface in the case where dwIPAddress == 0.

Return value

Returns 0 on success, any other integer value contains the error code.

Low level calling parameters

AH JOIN_GROUP (0x60)

DS:SI Pointer to GROUP_ADDR structure, documented in CAPI.H

Low level return parameters

If carry flag is set, AX = error code

LeaveGroup

The **LeaveGroup()** function causes SOCKETS to leave a multicast group.

C syntax

int **LeaveGroup(**DWORD *dwGroupAddress*, DWORD *dwIPAddress*);

Options

dwGroupAddress

The group address on which multicast datagrams are being received.

dwIPAddress

The IP address for the interface being used. The first interface to be specified in SOCKET.CFG is the default interface in the case where dwIPAddress == 0.

Return value

Returns 0 on success, any other integer value contains the error code.

Low level calling parameters

AH LEAVE_GROUP (0x61)

DS:SI Pointer to GROUP_ADDR structure, documented in CAPI.H

Low level return parameters

If carry flag is set, AX = error code

GetBusyFlag

The **GetBusyFlag** function returns the busy status of SOCKETS. **GetBusyFlag** is callable at a low level only; there is no high-level function.

Low level calling parameters

AX GET_BUSY_FLAG

Low level return parameters

ES:SI Pointer to the busy flag byte.

Examine only the four low-order bits. A non-zero value indicates that SOCKETS is currently busy. A value greater than 1 indicate that SOCKETS is not only busy, but is re-entered.

Error Codes

Error Value Error Code Meaning

NO_ERR 0 No error

ERR_IN_USE 1 A connection already exists

ERR_DOS 2 A DOS error occurred

ERR_NO_MEM 3 No memory to perform function

ERR_NOT_NET_CON 4 Connection does not exist

ERR_ILLEGAL_OP 5 Protocol or mode not supported

ERR_NO_HOST 7 No host address specified

ERR_TIMEOUT 13 The function timed out

ERR_HOST_UNKNOWN 14 Unknown host has been specified

ERR_BAD_ARG 18 Bad arguments

ERR_EOF 19 The connection has been closed by peer

ERR_RESET 20 The connection has been reset by peer

ERR_WOULD_BLOCK 21 Operation would block

ERR_UNBOUND 22 The descriptor has not been assigned

ERR_NO_SOCKET 23 No socket is available

ERR_BAD_SYS_CALL 24 Bad parameter in call

ERR_NOT_ESTAB 26 The connection has not been established

ERR_RE_ENTRY 27 The kernel is in use, try again later

ERR_TERMINATING 29 Kernel unloading

ERR_API_NOT_LOADED 50 SOCKETS kernel is not loaded

SOCKETS Programming Tutorial

Sample Application Examples

Compiler Notes

The attached examples are designed for use with Borland 5.2 compiler. A makefile (example.mak) has been provided for reference. These are real mode examples only, compiled as a 16-bit DOS application with small memory model. The module “compiler.h” can be ported for use with other compilers, currently it supports various Microsoft and Borland compilers. To use the makefile with BC 5.2 simply type:

```
Make –fexample.mak
```

Included Files

- CHAT.C
- CHAT.PT

- MCCHAT.C
- UDPCHAT.C
- UDPCHAT.PT
- CHAT.MAK
- CAPI.C
- CAPI.H
- COMPILER.H
- _CAPI.C

CHAT

Overview

A TCP based CHAT application. A server is started on the defined CHAT port. All connections made to this server, as well as those made by the local user to other servers, are put in a list. Whatever data the local user enters is sent to all the connections in this list (When the user hits Enter). Any data received from any of these listed connections is displayed on the screen. This program and its’ functions are single-threaded and non-reentrant and should be used as such.

Protocol

A CHAT server accepts TCP connections from several clients on port 5000. Once connected, a client may send lines of text to the server. Those lines are then sent out to all connected clients. The net result is that all connected users can see the typed lines of text from all other users.

Implementation

For the sake of simplicity, this implementation is not designed for portability. Since the Compatible API is only available for DOS-based stacks, the code relies on certain DOS features like keyboard hardware. Also, several basic functions are simply presented rather than explained.

Programming Style and Naming Conventions

Datalight strongly recommends the use of the Hungarian naming convention. The code in this tutorial relies on that convention. For those unfamiliar, Datalight recommends several reads of the Microsoft Press book, “Writing Solid Code” by Steve Maguire. Here are a few of the prefixes and a brief explanation:

i	integer
sz	string, terminated by zero
rg	range, an array of elements
c	character
p	pointer

UDPCHAT

Please review the differences between a tcp (stream) session and a udp session. When designing an application to use a udp session we eliminate many connection issues by simply not caring if the recipient has in fact received the packets being sent. 'CHAT' means to to talk and to listen. When using UDP, it means that for the talk size we just send a broadcast message on the lan and for the listen side we start a UDP server. This program and its functions are non-reentrant.

MCCHAT

A program that enables users to CHAT using multicast udp. 'CHAT' means to to talk and to listen. When using UDP, it means that for the talk side we just send a multicast message on the LAN and for the listen side we receive on the same UDP socket. This program and its functions are non-reentrant. The changes between udpCHAT and MCCHAT are minimal. We add the group CAPI calls, JoinGroup() and LeaveGroup()and remove the ListenSocket()and GetPerrAddress() calls. We add an include for “ctype.h”.