Datalight SOCKETS™

Developer's Guide

Distributed by RTD Embedded Technologies
SWM-640070001
rev A

Datalight **SOCKETS**™ **Developer's Guide**

Contents

Chapter 1, Introduction

About SOCKETS

Datalight SOCKETS is a command-driven Internet protocol software package for embedded systems running DOS. Datalight SOCKETS provides a powerful data communication facility whereby embedded systems and users of embedded systems can communicate with other computers (including PCs and mainframes) and their printers.

Datalight SOCKETS is a networking communications application, providing both client and server services on the network, which allows you to:

- Run applications on a TCP/IP host system from a remote embedded system.
- Transfer data between an embedded system and TCP/IP hosts.
- Run network aware applications on an embedded system.
- Print to an embedded system from TCP/IP hosts and vice versa.

Getting Started

The installation and configuration of Datalight SOCKETS may differ significantly from that of other similar products. Please take your time and do not progress beyond any step until you are confident that you understand it. If you find a problem that is not documented, please call our technical support department for assistance rather than continuing to the next step.

The following steps are provided to help you create a fully internet-aware target system in the shortest time possible.

1. Familiarize yourself with the SOCKETS product by browsing this chapter.
2. Decide whether you will be testing your applications on your development machine or on the target machine. You may reference the startup guide in Chapter 2 for easy setup information. Please note that the configuration of SOCKETS on your development system may differ from the configuration on your target hardware.
3. Install the SOCKETS package as described in Chapter 4.
4. Refer to Chapter 5, SOCKETS Configuration and Use. Continue to the following chapter for a full configuration reference.
5. Test your configuration with XPING before continuing. Proper configuration of SOCKETS is critical! Do not proceed until you can successfully ping a host on your LAN.
6. If you are having general network or performance problems, refer to Chapter 11.

References

The formal network standards for the TCP/IP protocol suite is available as a set of documents known as Requests for Comments (RFCs).

Full specifications for IP are given in:

- ARPA RFC-791
- MIL-STD-1777

Specifications for TCP are given in:

- ARPA RFC-793
- MIL-STD-1778

Specifications for FTP are given in:

- ARPA RFC-959

SOCKETS General Characteristics

System Requirements

SOCKETS requires an IBM compatible 186 or higher system with a minimum:

- 512KB of RAM
- ROM-DOS 6.22, ROM-DOS 7.1, or MS DOS 6.22

Network requirements for SOCKETS requires one or more of the following, depending on the network configuration:

- Ethernet
- Token Ring
- Any network supporting Novell NetWare
- Asynchronous Point-to-Point link (PPP, SLIP, or CSLIP)
- Any network supporting the packet driver specification
- Any network supporting the NDIS specification
- Any network supporting the ODI specification

Application Program Interface (API)

An API allows applications to request services and pass data to and from the TCP/IP stack, thereby enabling third-party developers and end users to develop their own applications running on SOCKETS. SOCKETS offers a standard API that supports stream services (TCP), datagram services (UDP) and ICMP echo request/reply.

SOCKETS are the interface points through which data is sent and received between the processes of various host systems.

Compatible API

SOCKETS contains an API that is compatible with a wide range of third party TCP/IP applications. Chapter 8 contains a reference for the functions that comprise the compatible API. Refer also to the file CAPI.H.

SOCKETS API

SOCKETS offers a socket interface for applications requiring a TCP/IP stack in the DOS environment. Such applications include terminal emulators, database clients, and so on. The file API.H contains function description for the SOCKETS API.

About TCP/IP

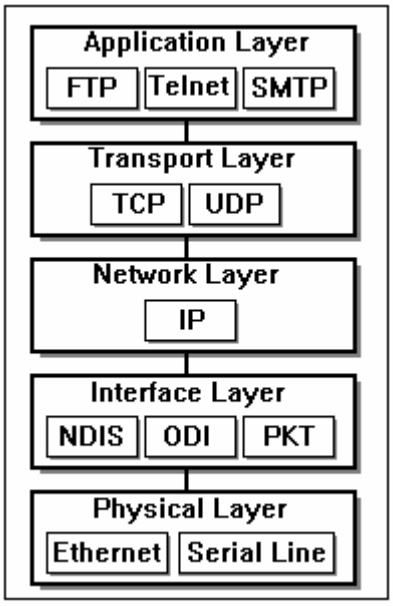
The following sections contain a general description of TCP/IP and provide an introduction to the operation of TCP (Transport Control Protocol) and IP (Internet Protocol) and its components.

Transmission Control Protocol and Internet Protocol, collectively known as TCP/IP, comprise a set of computer data-communication conventions or protocols. These protocols were developed by major users of computer based equipment, principally the U.S. Department of Defense, so that the equipment supplied by different manufacturers could exchange data and information. TCP and IP are only two of the major protocols in a system consisting of many protocols.

TCP/IP Layers

A TCP/IP implementation consists of a series of software layers, where each layer performs specific functions for the layer above and below it. TCP/IP uses four software layers and one physical layer, as follows:

- The Application Layer selects the appropriate service for applications.
- The Transport Layer provides end to end data integrity.
- The Network Layer switches and routes information.
- The Interface Layer transfers units of information to the physical layer.
- The Physical Layer provides transmission onto the network.



Media Support

Various methods of media are available for TCP/IP communication. A brief description of the supported media types follows.

Ethernet and Token Ring

The packet driver, NDIS and ODI standards are supported.

Serial Line IP (SLIP)

SLIP uses standard asynchronous lines to transfer IP datagrams. The SLIP provided by SOCKETS is compatible with that used on UNIX systems. Error checking is provided by checksums that are part of IP, TCP and UDP.

Serial Port 16550 UART

Where serial ports are used, SOCKETS checks for and uses FIFO buffered 16550 UARTs for faster throughput. It is strongly recommended that buffered UARTs be used for high-speed applications.

Flow Control

The following modes of flow control are supported:

No flow control:

None - always send

Software flow control:

XonXon/Xoff uses Ctrl-Q/Ctrl-S to start/stop data flow

Hardware flow control:

CTS - clear to send
RTS - request to send
DCD - data carrier detect
DSR - data set ready
DTR - data terminal ready

Inverse hardware flow control:

ICTS, IRTS, IDCD, IDSR, or IDTR.

Compressed Serial Line IP (CSLIP)

CSLIP is an enhancement of SLIP by implementing Van Jacobson header compression. CSLIP uses more memory than SLIP but provides better throughput and faster response times, especially on small packets.

Point to Point Protocol (PPP)

The SOCKETS PPP (Point to Point Protocol) implementation uses any asynchronous port with a non-shared interrupt and can support dial-up operation. LCP, PAP, CHAP and IPCP are implemented and can be configured to act as a server or as a client to access any server accepting PPP connections such as the RAS services of NT Server . For information regarding the definition of an interface for and setup of PPP, refer to "Setting up a Modem" on page 19.

Routing Support

There are various protocols within the implementation of a TCP/IP stack that occur below the Transport Layer. These protocols can be used to set specific packet routing information or to allow the TCP/IP stack to be dynamically configured. A brief description of these protocols follows.

Bootstrap Protocol (BOOTP)

BOOTP is a UDP/IP based protocol that provides a means to assign an IP address to a booting host dynamically and without user supervision. BOOTP can also supply the net mask, host name, and address of a domain name server. One obvious advantage of this procedure is the centralized management of network addresses, which eliminates the need for per-host unique configuration files. SOCKETS implements the BOOTP client whenever it is started with no (or the 0.0.0.0) IP address supplied.

For file server based networks, SOCKETS workstations can be configured in a similar way using the SETHOST utility. For further information on SETHOST, refer to "Chapter 10:Managing the Network and Troubleshooting."

Dynamic Host Configuration Protocol (DHCP)

DHCP is a UDP/IP based protocol that provides a means to assign the IP address dynamically to a booting host and without user supervision. It can also supply the net mask, host name, address of a domain name server, and other parameters. An advantage of this procedure is the centralized management of network addresses, which eliminates the need for per-host unique configuration files. SOCKETS implements the DHCP client whenever it is started with the 0.0.0.1 IP address supplied. All LAN interfaces specified when this IP address is in use will attempt to use DHCP to resolve the IP address, the subnet mask, hostname, default router and DNS server(s).

Address Resolution Protocol (ARP)

ARP provides mechanisms for hosts to search and find the MAC (Ethernet) addresses of other hosts on the network. SOCKETS supports ARP for Ethernet and Token Ring controllers.

Proxy ARP is utilized for gateways. A route can be designated as supporting proxy ARP. When a gateway receives an ARP request for a host and it has a route to reach that host, it responds to the ARP request. For further information on Proxy ARP refer to "Chapter 6, SOCKETS Configuration Reference."

Note: Proxy ARP should be used with care and not in conjunction with RIP. If more than one host responds to an ARP request it may cause system problems.

Routing Information Protocol (RIP)

The Routing Information Protocol allows a SOCKETS gateway to advertise routes, and allows SOCKETS to recognize advertised routes from SOCKETS and other gateways. Both versions 1 and 2 are supported. For further information on RIP, refer to Chapter 6 “SOCKETS Configuration Reference”.

Alternate Routes

SOCKETS can be set up to provide more than one route to a destination network. The first route is used until SOCKETS detects that the route is no longer functioning, in which case the next route in the list is used until the first becomes available again. SOCKETS determines the functioning of a route by performing an ICMP echo request (or ping) to a designated host.

Simple Network Management Protocol (SNMP)

The Simple Network Management Protocol is an automated tool used by administrators to monitor and control TCP/IP based networks. It allows you to edit networking information maintained by the hosts and routers attached to the network. SNMP operates over the User Datagram Protocol (UDP) which provides a connectionless service for exchanging messages while avoiding the overhead (and reliability) of TCP.

About the Data Transmission Protocols

File Transfer Protocol (FTP)

The FTP protocol transfers binary (image) and/or text (ASCII) files between host systems. FTP uses two TCP connections: one for exchanging commands and responses in the form of ASCII strings, the other for the actual data transfers. FTP is implemented in two parts, the server and the client. The server supports multiple simultaneous remote users, while the client provides an interactive or batch interface for performing remote file and directory maintenance and file transfers.

A user prompt to specify a name and password that have been registered/configured on the other computer controls file security. Provision is made for handling the transfer of files between machines with different character sets, end-of-line conventions, and so on.

Unlike network file system protocols, the FTP utility only transfers files between systems.

Telnet

Telnet and other terminal emulators are available from Datalight but not included within the SDK.

Telnet (Network Terminal Protocol) allows users to log in on any other host connected to the network. Specifying the host with which a connection is required starts these remote sessions. Once a connection is established, any local keyboard input is relayed to the remote host and any terminal output from the remote is displayed on the local screen. This is much like a dial-up connection in that the remote system requires log-in and password procedures, as you would encounter in dial-up systems.

At the end of a remote session a logoff command exits the telnet program, and returns the user to the local computer.

Simple Mail Transfer Protocol (SMTP)

The Simple Mail Transfer Protocol (SMTP) allows electronic messages to be sent between hosts on the network. The SMTP server receives mail while the SMTP client sends mail.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) is a second-generation, connection-oriented protocol that corresponds to the Transport layer protocols described in OSI. TCP forms a connection between the workstation and the system with which it intends to communicate. In a network, several systems can communicate across the same network cabling or the same gateways. To ensure that each transmission shares the transmission media equally, transmitted data are broken into manageable pieces known as segments.

TCP is responsible for:

- Breaking data into the appropriate segments.
- Numbering the segments sequentially before sending them.
- Reassembling and verifying the segments at the destination.

The sequential numbers are used to reassemble the segments at the destination. To assist in this procedure, TCP places a header at the beginning of each segment. The header contains the Source Port, the Destination Port, the Sequence number, and a checksum.

A checksum is a mathematical computation of the octets in the segment before it is sent. The same computation is performed at the destination to verify the integrity of the segment data. If the checksum results match, an acknowledgement is sent from the destination to the source. If the checksums do not match, the segment is discarded without an acknowledgement being sent and the source retransmits the segment.

User Datagram Protocol (UDP)

UDP provides an unsequenced, unreliable, connectionless transport service. It can act as an alternative to TCP for applications that do not require the same amount of control. The Domain Name protocol, Routing Information protocol and the Simple Network Management protocol all make use of UDP.

Internet Protocol (IP)

The Internet Protocol is a connectionless network layer protocol. It was designed to handle a large number of inter-network connections, for both LAN and WAN applications. The IP implementation basically addresses and sends the segments. IP relies on the IP address to deliver and receive segments.

The IP address is a 32-bit address assigned to a TCP/IP node. The IP address of each node must be unique on a network. No two nodes anywhere on the network can have the same IP address.

32-bit addresses are generally represented with decimal notation, which separates the four bytes of the address with periods. A typical IP address conversion is as follows:

Type of Format	Example of IP Address
32-bit Binary Format	10000000 01100101 01100110 01100111
Hexadecimal Format	80 65 66 67
Decimal Format	128 101 102 103
Decimal Notation	128.101.102.103

Although the IP address is represented as a single value, it contains two pieces of information:

- The first part of the address is your network identification. All machines on the same network have the same prefix.
- The second part of the address is your host identification. No two nodes share the same suffix.

Internet addresses fall into three major addressing classes. The address class that you request should be based on the maximum number of network nodes in your system.

Class	IP Address Range	No of Local Nodes
A	0.0.0.1 to 127.255.255.254	1-16,777,214
B	128.0.0.1 to 191.255.255.254	1-65,534
C	192.0.0.1 to 223.255.255.254	1-254

The table above shows the three Internet address classes with their associated IP address range and the number of local nodes possible per class.

Internet Control Message Protocol (ICMP)

ICMP is used for IP error control and diagnostic and provides error messages such as:

- *Destination unreachable*
- *Time to live (TTL) expired*
- *Header problems*

ICMP also support echo request and echo reply, better known as a ping operation to test a host for connectivity and response time.

HTTP Protocol

Hypertext Transfer Protocol is the backbone for the World Wide Web. SOCKETS provides HTTP functionality through an embedded web server and various client applications. For further information please see Chapter 7 “SOCKETS Applications”. Web enabling your device with Datalight SOCKETS will allow easy control of embedded devices from standard desktop web browsers

Chapter 2, Startup Guide

SDK Overview

Upon receiving SOCKETS, the first step involved is installing to a development system. The install process will create, by default, a directory of C:\DL\SOCKETS.

The files installed cover:

- Kernel applications (the core of TCP/IP communication)
- Configuration examples
- Utilities
- Optional applications
- API with programming examples

The remainder of this chapter describes how to proceed after SOCKETS has been installed to the development system.

Installation

Insert the SOCKETS disc into your CD-ROM and run "D:\INSTALL.EXE" where "d" is the appropriate drive letter for your CD. The install process will create a C:\DL\SOCKETS directory. Within the directory C:\DL\SOCKETS will exist the kernel applications SOCKETM.EXE and SOCKETP.EXE. Beyond that, there are subdirectories for:

- UTILS - contains tools for troubleshooting and configuring SOCKETS.
- CONFIGS - contains example configuration files for SOCKETS connections.
- EXAMPLES - contains programming examples.
- CAPI - contains the APIs necessary for linking your application to SOCKETS.
- CLIENTS - contains the client applications.
- SERVER - contains the SOCKETS server providing HTTP and FTP services.

Environment Variables

SOCKETS uses environment variables to determine the location of necessary configuration files. They can be set, using DOS *SET* command, within the autoexec.bat file at startup or within a batch file prior to SOCKETS being loaded. They are:

- SOCKETS
- HTTP_DIR
- HOSTNAME

- FTPDIR

Examples

Set SOCKETS=C:\NETWORK

Set HTTP_DIR=C:\HTML

Set HOSTNAME=FTPDEMO

Set FTPDIR=C:\FTP

Remarks

The environment variable *SOCKETS* is used to indicate to the SOCKETS kernel where configuration files, license file for the demo version, and access rights files are located. For example "Set SOCKETS=C:\NETWORK" would cause SOCKETS to look for configuration files within the directory of C:\NETWORK.

The environment variable *HTTP_DIR* is used by the SOCKETS server and indicates the location of html files if the files are stored in a separate directory from the server executable. To continue with the previous example if the server were in the directory of C:\NETWORK and the html files were in C:\HTML then the environment variable would read "SET HTTP_DIR=C:\HTML".

The environment variable *HOSTNAME* is used by SOCKETS to indicate the banner name displayed during an FTP session.

The environment variable *FTPDIR* is used by SOCKETS to indicate the location of the temporary file created by the SOCKETS server during an FTP session. Please refer to Chapter 7 for more details.

File Selection

What you wish to do with the SOCKETS TCP stack will dictate which files are to be transferred to the target hardware. At a minimum your target hardware will require the TCP stack executable:

- SOCKETM.EXE for serial / ppp connection
- SOCKETP.EXE for an Ethernet connection
- SOCKET.CFG for both serial and ethernet use
- MODEM.MCF for a serial / ppp connection

If you wish to use the Datalight Web Server with SOCKETS you will need the HTTPD.exe file located within C:\DL\SOCKETS\SERVER. You will also need an index.htm file that dictates what information the web server displays. One has been provided within the same directory. Full explanations of the files are available in Chapters 5 and 6.

Configuration

The configuration of SOCKETS is determined by what you would like to do with the TCP/IP stack. Some examples are provided within Chapter 5 of the types of configurations easily accomplished with SOCKETS. For quick testing of SOCKETS please reference the sample configuration files supplied within the \CONFIGS directory with their explanation in Chapter 5.

To make your own configuration, run the supplied file sconfig.exe to configure SOCKETS for your target hardware. SCONFIG.EXE is a text-based tool that will prompt for various settings necessary to customize SOCKETS. At completion sconfig.exe creates the file SOCKET.CFG which must accompany SOCKETS to the target hardware. Running SCONFIG may take place on either your target hardware or your development system. If you wish to configure the stack after transferring files to your target hardware, please include the file SCONFIG.EXE in your list of files to be transferred to your target hardware.

Transfer

Transfer the selected files onto the target system into a \SOCKETS\ directory. As Datalight ROMDOS has various methods of serial port transfers available, please refer to the ROMDOS manual for specific details.

Testing

If you are using an Ethernet TCP stack you must first load a packet driver specific to your NIC. The packet driver provides a software interrupt to allow communication between the NIC and SOCKETS (default is 0x60). If you do not have a packet driver or are unsure of its use please contact Datalight Technical Support. Although Datalight does not manufacture packet drivers our support team is knowledgeable about packet driver implementation.

Type SOCKETM.EXE or SOCKETP.EXE to launch DL SOCKETS. The various options for launching SOCKETS are:

/n=normal_sockets
/i=capi_interrupt
/d=dos_compatible_sockets
/m=memory_size
/p=print_delay
/s=stack_size
/v=interrupt_vector
/q
/0
/u

Once SOCKETS has been loaded correctly, the most basic test to ensure that everything is working correctly is to “XPING” a known server or gateway that the SOCKETS machine is connected to. If the ping is returned everything is working, if the ping fails please refer to the troubleshooting section of the SOCKETS manual.

Custom Applications

The key to designing an application to work with DL SOCKETS is the CAPI. The CAPI provides the interface between your application and DL SOCKETS. The CAPI must be linked into your application at compile time. At the time of this manual the CAPI has been designed to work with Borland and Microsoft C 1.52 compilers. Other ports of the CAPI will be posted upon completion. The SOCKETS manual details each of the functions within the CAPI and can be used as a reference if you need to make changes in order to compile the CAPI into your application. Please contact your sales rep or technical support for any updated compiler compatibility list.

Chapter 3, Physical & Interface Layers

Setting up Network Drivers

Overview

SOCKETS provides support for a wide range of LAN adapters supporting the Packet driver specifications.

The packet driver specification is adhered to by many vendors of LAN adapters and is also freely available as public domain software from CompuServe and other sources. SOCKETS only supports the packet driver specification, but public domain converters or "shims" are available to convert from NDIS or ODI to packet driver. Some "shims" may be obtained from Datalight Inc. but are not distributed with SOCKETS. SOCKETS may use up to eight packet drivers.

Using the same network board, SOCKETS can operate simultaneously with other network operating systems such as Novell, Microsoft, Banyan and others.

Note: Software configuration of network hardware, such as Plug and Play settings or packet drivers, must be completed before SOCKETS is loaded. Failure to properly load and configure the appropriate software results in error messages from SOCKETS. To help you isolate which device is failing; the [Interface] entry is referenced in those error messages.

The supplied utility PDTEST.EXE can be used to test or diagnose your packet driver before attempting to start SOCKETS. For further information on PDTEST, refer to Chapter 11 "Managing the Network and Troubleshooting."

ODI Driver Installation

If you already have Novell using ODI installed, just modify AUTOEXEC.BAT and the existing NET.CFG, otherwise make a \ODI directory on your hard drive and copy the following files to it:

Filename	Location
LSL.COM	Novell WSGEN floppy disk
NETX.EXE or VLM.EXE	Novell WSGEN floppy disk
IPXODI.COM	Novell WSGEN floppy disk
ODIPKT.COM	NIC Driver disk *
NIC ODI Driver	NIC Driver disk

*** Note:** ODIPKT.COM may not be provided by your vendor. In that case, there are many solutions available on the Web – just do a search for "odipkt.com".

Examples of ODI Drivers:

SMC8000.COM, SMCPLUS.COM, NE2000.COM, 3C5X9.COM or 3C503.COM

If you do not have a NET.CFG file, create an ASCII text file in the \ODI directory called NET.CFG that should look similar to the following:

```
Link Support
   buffers 8 1600
Protocol IPX
   Bind ODI_driver
Link Driver ODI_driver
   int irq
   port io port
   frame ETHERNET_802.3
   frame ETHERNET_II
```

The important parts to check are the buffers in the Link Support section and the order of the Ethernet Frame (also called Envelope Type) lines. Each Frame line specifies a logical board, starting from number 0. ODIPKT should link to the Ethernet_II board, which, in this example, would be board number 1. The file must contain at least the following:

```
Link Support
   buffers 8 1600
Link Driver ODI_driver
   frame ETHERNET_II
```

Example for SMC:

```
Link Support
   buffers 8 1600
Link Driver SMCPLUS
   Port #1 280
   mem #1 000D0000 2000/10
   Int #1 3
   frame ETHERNET_802.3
   frame ETHERNET_II
```

Add the following lines to your AUTOEXEC.BAT file:

```
CD\ODI
LSL
rem Your NIC ODI driver
ODIPKT 1 96
IPXODI
NETX
F:
```

The syntax for ODIPKT.COM is:

ODIPKT logical_board interrupt_vector

The interrupt vector must be specified in decimal; for example 0x60 = 96.

logical_board is the index of the Frame type entry starting at 0. The normal frame type to use with SOCKETS on Ethernet is ETHERNET_II which is the second entry (or logical board 1) in the preceding example.

Example

```
CD\ODI
LSL
```

```
SMCPLUS
ODIPKT 1 96
```

Using a Memory Manager with an ODI Driver

If a board using mapped memory (for example, SMC) is used with an upper memory block manager (for example, EMM386), the shared memory must be excluded as follows:

```
DEVICE = C:\DOS\EMM386.EXE X=D000-D3FF
```

Packet Driver Installation

Instead of an ODI (Novell) driver, you can use the packet driver for your network board. After you have installed SOCKETS, make a \PKTDRV directory on your hard disk and copy your NIC (Network Interface Controller) packet driver, from the NIC Driver disk. The various manufacturers supply their own packet drivers that may differ from what is documented here. Always consult the software and documentation supplied with your network board first.

You can perform development work exclusively on your development machine and integrate your program on the target machine at a later date. In this case, Datalight recommends you purchase Dan Lanciani's Virtual Packet Driver for Windows from http://www.danlan.com. When installed, SOCKETS coexists with the native Win32 TCP/IP stack without conflict, *as long as different IP addresses are used for the two stacks*.

Note: Many PCI packet drivers require no command line parameters to load and, by default, set up an interrupt vector of 0x60.

Packet Driver Filename	Manufacturer
PKT8000.COM	SMC/WD
3C503.COM	3Com
3C505.COM	3Com
3C507.COM	3Com
NWPD.COM	Accton

Choosing a Packet Driver – SMC Cards

If the system is running on a Novell network run your normal IPX.COM file and then run 8003PKDR.EXE without any parameters. The packet driver picks up its settings from the IPX.COM file. If you are not running on a Novell network you must run 8003PKDR.EXE with certain of the following parameters:

Parameter	Default Value
/S: (slot#)	0(none)

Parameter	Default Value
/R: (ram base)	D000
/B: (io base)	0(none)
/I: (irq)	3
/E: (class 1 int)	60(hexadecimal)
/8: (class 11 int)	00(none)
/M: (mx multicast address	8

For example if you are running an ISA bus machine with NIC set at io base=280, ram base=D000 and irq=3, then your command line would be:

```
8003PKDR /R:D000 /B:280 /I:3 /E:60
```

Choosing a Packet Driver – 3Com Cards

If you are running on a Novell network with these packet drivers you must generate a special IPX.COM file to run over the packet driver. All 3Com packet drivers work in the same manner.

Switch	Description
-n	Converts the IPX packets between the Ethernet II type 8137 encapsulation used by BYU's PDSHELL IPX interface code and the 802.3 style encapsulation normally used on Ethernet by NetWare servers, shell and boot PROMs. Also converts incoming type 8137 packets to type 8138 for NetWare compatibility.
-d	Delays the initialization of the Ethernet board until the packet is used for the first time.

Usage:

packet_driver_name [-n] [-d] [-w] packet_int_no [int_no[io_addr]]

The 3C503 driver requires an additional parameter; the cable type - 0 for thick and 1 for thin.

Choosing a Packet Driver – Accton Cards

If you are using a Novell network with these packet drivers you must generate a special IPX.COM file to run over the packet driver.

Use the -c option only if your board is not an Accton board. You can use this option to run the packet driver on another manufacturer's board. Before using this option, ensure that your board is compatible with NE1000, NE2000, NE2 or WD.

Switch	Description
-c	Use this switch only if your board is not an Accton board. You can use this option to run the packet driver on another manufacturer's board. Before you use this option, make sure your

Switch	Description
	board is compatible with NE 1000, NE2000, NE2, or WD.
-n	Converts the IPX packets between the Ethernet II type 8137 encapsulation used by BYU's PDSHELL IPX interface code and the 802.3 style encapsulation normally used on Ethernet by NetWare servers, shell and boot PROMs. Also converts incoming type 8137 packets to type 8138 for NetWare compatibility.
-d	Delays the initialization of the Ethernet board until the packet is used for the first time.
-u	Unloads the packet driver.
-h	Displays the on-line Help

Usage:

NWPD[-cTYPE] [-d] [-n] [-w] [-u] [-h] *packet_int_no int_no* [*io_addr*][*mem base*]

Using a Memory Manager with a Packet Driver

If a board using mapped memory (for example, SMC) is used with an upper memory block manager (EMM386), the shared memory must be excluded:

```
DEVICE=C:\DOS\EMM386.EXE X=D000-D3FF
```

Setting up a Modem

While this section documents only simple PPP options in SOCKETS, there are more advanced options and additional protocols such as SLIP and CSLIP.

Modem Definition File Syntax

The format of the modem definition file specifies commands entered on separate lines starting with one of the following characters in the first column:

i	***initialisation string/script*** (for the modem)
n	***telephone number*** to dial (one number per line)
r	***retry_count*** (when a connection or script failed)
x	***exchange_id*** (XID for user identification)
d	***dial command/script*** (talk to modem till connected)
a	***answer prompt script*** (to remote for him to login)
c	***connect string/script*** (to remote for me to login)
p	***parameters*** (for debugging/watching)
b	***command_character*** (default is @ in this syntax)
#	***comments*** (what all good programmers do)

The *initialisation* ***strings***, ***answer prompt***, ***connect string*** and ***dial commands*** consist of modem and login commands or prompts and special functions to cause delays and wait for DCD or strings. The simple scripts are one line strings where commands start with the ***command_character*** (default is @) followed by a script command, followed by a time in

milliseconds (and a receive search string in the case of **@r**). The following script commands can be used:

@w ***time***	wait for the full ***time*** specified
@d ***time***	wait for DCD (modem to get connected)
@f ***time***	wait for ^F XID ^M and find **mdd** with the matching XID
@a ***time***	wait for IP Address (any where in data stream) and use it as your own for this interface
@r ***time string***@	wait for ***time*** to receive ***string***
time@***.string***@	terminate ***time*** if followed by a string starting with digits
string	send ***string*** to modem/remote (default in all scripts)
@n	insert telephone number (used in dial string)
@x	insert XID (used in connect string)
@@	send ***command_character*** (@) to modem/remote

Note:

- The ***time*** is given in milliseconds and is terminated by the first character that is not a decimal digit. Maximum ***time*** is one hour (3 600 000ms).
- The script is aborted when a conditional wait times out, without making the connection.
- Receive strings (**@r**) are terminated with a ***command_character*** (@).
- Send strings are terminated with the next command or the end of a line. A carriage return character (CR) is not automatically added.
- To put control characters in the string use ^***n*** where ***n*** is A for 0x01, B for 0x02 and so on. A CR is ^M and a LF is ^J. Use ^***SPACE*** to send the ^ character to the modem.
- Do NOT include spaces as separation characters since all characters are interpreted.
- The @'s are strictly interpreted from left to right - do not confuse them with terminating and initiating @'s. After **@r,** a terminating @ must follow. An initiating @ for the next command MUST be given - it is not automatically assumed since a send string is implied by default.

Example

`@w1000`	wait for 1 second
`@d30000`	wait for DCD, retry dial after 30 seconds.
`@@`	send @ to modem/remote
atdt`@n^M^J`	dial the next number in the list
`@r2000login@`	wait for the string "login"

parameters comprise an optional string specifying whether dial communications must be displayed. Use **x** to display transmit data and **r** for received data. The display is only active while DCD is low.

An example or two is the fastest way to understand the explanations above. For a simple connection where no passwords or identity checks are needed, try the following:

```
# Modem definitions for Zoltrix Hayes compatible modem
# This example is for 'asy' interfaces making outgoing
# connections with no logon sequence.
```

```
#
# initialisation string
# (Warning: consult the manual for your own modem)
# send "a", wait half a second, send "a", wait half a second,
# send "ats0=0" - do not use auto answer
# send "&c1" - use state of carrier for DCD
# send "&d3" - disconnect when DTR low
# send "&k3" - enable RTS/CTS flow control
# send "&q5" - select error correction
# send "<CR><LF>", wait 100 milliseconds
i a@w500a@w500ats0=0&c1&d3&k3&q5^M^J@w100
#
# dial command - send "<CR><LF>", wait 2 seconds
# send "atdt<number><CR><LF>"
# wait 30 seconds for DCD
d ^M^J@w2000atdt@n^M^J@d30000
#
# two numbers to dial in rotation
n 790-1234
n 790-1235
#
# parameters: r=show modem receive, x=show modem xmit
p r
#
# number of retries
r 5
```

A more typical login sequence (where a user ID and password is required) will use the connect script as in this example:

```
# modem definitions for US Robotics 28.8 Hayes compatible modem
# send "a", wait a tenth second, send another "a", wait a tenth
second,
# send "at&c1" - use state of carrier for DCD
# send "&d3" - disconnect when DTR low
# send "&i0&r2&h1" - enable RTS/CTS flow control
# send "&m4" - select error correction
# send "s0=0" - do not use auto answer
# send "<CR><LF>", wait 100 milliseconds
i a@w100a@w100at&c1&d3&i0&r2&h1&m4s0=0^M^J@w100
# number to dial
n 03456789
# number of retries
r 4
#
# dial command - send "<CR><LF>", wait 1 second
# send "atdt<number><CR><LF>"
# wait 40 seconds for DCD
d ^M^J@w1000atdt@n^M^J@d40000
```

```
#
# connect script - wait 7 seconds for login prompt
# send user ID, wait 2 seconds for password prompt
# send password, wait 2 second for acknowledgement
c @r7000ogin@myuserid^M@r2000sword@mypassw^M @r2000Welcome@
# parameters: r=show modem receive, x=show modem xmit
p rx
```

Note that the '@m's in the above connect string are not commands since the @ is interpreted as the end of a **@r** 'wait for receive' string. Sometimes you would have two @'s like this example: Should you want to put a small delay before replying to a prompt it would contain @@ as follows:

```
c @r7000ogin@@w100myuserid^M
```

For receiving incoming calls, the answer command/script is used. You can use it to do a single user ID/password controlled login, just a user ID login, or unconditional acceptance. The more general case is to use **mdd** interfaces with an XID (exchange ID) to validate multiple users. Examples for using Multi Destination Drivers (MDD) are given below. If your SOCKETS workstation accepts only incoming calls, you do not need to enter dial commands.

Retry Strategy on Time-out

For outgoing calls, when a wait for DCD (@d) times out without receiving DCD, SOCKETS will drop the call (by lowering DTR). If the number of retries has not been exhausted, SOCKETS will retry the dial command with the next phone number rotating through the list of numbers. If a connection with the remote modem is successful (received a DCD) and there is a timeout on the wait for ***string*** command in the connect script, the connect script is re-started for the number of retries specified. If the number of retries has been exhausted, SOCKETS will retry the dial and connect commands only after being prompted again by receiving traffic for this destination host or net.

For incoming calls, when there is a time out on the wait for XID or ***string*** commands, the answer-prompt script is re-started for the number of retries specified. If the number of retries has been exhausted, SOCKETS will drop the call (by lowering DTR).

Multi Destination Drivers

When you have more than one destination to or from dial-up links using SLIP, and are using more than one modem, you need a mechanism to link the logical interface (with IP address and routing info) to the physical interface (COM port with modem definitions). Using an **asy** type interface permanently links the logical and physical interface with one IP address. This works OK for dialling out to multiple destinations only where you are the end user and nobody routes through you.

Two interface types - mdd and **aslink -** have been defined to enable dial-up operation to other networks using SLIP.

- **mdd** defines a logical interface with associated IP address, routes, MTU, buffer limit and a modem definition file but without a physical interface. For each dial-up destination a **mdd** interface is created, and a route to each destination specified.

- **aslink** interface defines an uncommitted asynchronous interface with an associated COM port as specified by the I/O address, and interrupt vector. There isn't a modem file associated with an **aslink** interface.

When initial traffic is to be sent to a destination through a **mdd** interface, SOCKETS scans all **aslink** interfaces for the first available one and assigns it to the **mdd** interface. This association is broken when the dial connection is broken or when the dial attempt fails. A dial attempt fails when the retry count specified in the modem file expires.

Incoming calls can be received on any **aslink** interface. This interface must then be associated with the correct **mdd** interface (and IP address with route) for the calling host. An exchange identifier (XID) defined by both the calling host and the called host achieve this.

The protocol for exchanging the XID is implemented partly by SOCKETS and partly by the user defined 'connect' and 'answer' scripts in the modem definition files. When an incoming call is received by an **aslink** interface, as seen when DCD is raised, the 'answer' script is executed. This script must contain a command to wait for the XID and match it to a local **mdd** containing the same XID and an acknowledgement to the sender that the exchange has been successful. The XID *must* be preceded by an ACK (^F) and followed by a CR (^M). The 'connect' script of the calling host is used for this purpose. The following ‘answer’ and 'connect' scripts may be used:

```
a @w100^M^JYour ID?@f1000ccc
```

This means: Wait for 100 milliseconds (after receiving the DCD) send a CR/LF to write on a new line and prompt the user with 'Your ID?’ Wait 1 second to receive a valid XID matching with an **mdd** interface and then send a few 'c's to acknowledge 'connected' for the caller.

```
c @r500ID?@^F@x^M@r500c@
```

This means: Wait half a second to receive the 'ID?' prompt, send the required ACK (^F), XID (@x) and CR (^M). Wait another half second for acknowledgement from the remote with a 'c'.

It can occur that a request for making an outward connection and an attempt by an incoming connection clash at the same **mdd** interface. One example is when a connection is broken and both sides redial each other to restore the connection. A mechanism must be used to allow only one of the attempts to be successful. Therefore a **mdd** interface will drop an **aslink** trying to make an outgoing connection as soon as it senses another **aslink** trying to connect to it with an incoming connection.

In summary the modem commands used by the **mdd** and the **aslink** in their respective modem files for the following functions are:

- For initiating the modem: Always use the '**i**-' command in the **aslink** modem file.

- For making a call: Use the telephone numbers from the **n**-command in the **mdd** modem file, select any available **aslink**, and use its **d**-command. When the modems are connected, use the **c**-command in the **mdd** modem file to logon with its XID (**x**-command).
- For answering a call: Use the **a**-command in the **aslink** modem file to query the user for his XID (logon sequence), find the **mdd** modem file with the matching XID (**x**-command).

More examples are available in the HAYES.MOD file. See the command descriptions in the **Modem Definition File Syntax** section above.

Example

An **mdd** modem definition file for both incoming and outgoing connections:

```
n 790-1234
n 790-1235
x id-abc
c @r20000ID?@^F@x^M@r500c@
r 5
```

An **aslink** modem definition file for both incoming and outgoing connections:

```
i a@w500a@w500ats0=1&c1&d3&k3&q5^M^J@w100
a ^M^JYour ID?@f5000@w200ccc
d ^M^J@w2000atdt@n^M^J@d30000
r 5
```

Example

An **aslink** modem definition file for incoming connections:

```
i a@w500a@w500ats0=1&c1&d3&k3&q5^M^J@w100
a ^M^JYour ID?@f5000@w200ccc
r 5
```

An **mdd** modem definition file for incoming connections:

```
x id-abc
```

Example

An **mdd** modem definition file for outgoing connections:

```
n 790-1234
n 790-1235
x id-abc
c @r20000ID?@^F@x^M@r500c@
r 5
```

An **aslink** modem definition file for outgoing connections:

```
i a@w500a@w500ats0=0&c1&d3&k3&q5^M^J@w100
d ^M^J@w2000atdt@n^M^J@d30000
r 5
```

Chapter 4, TCP/IP Basics (Network Layer)

Setting up SOCKETS

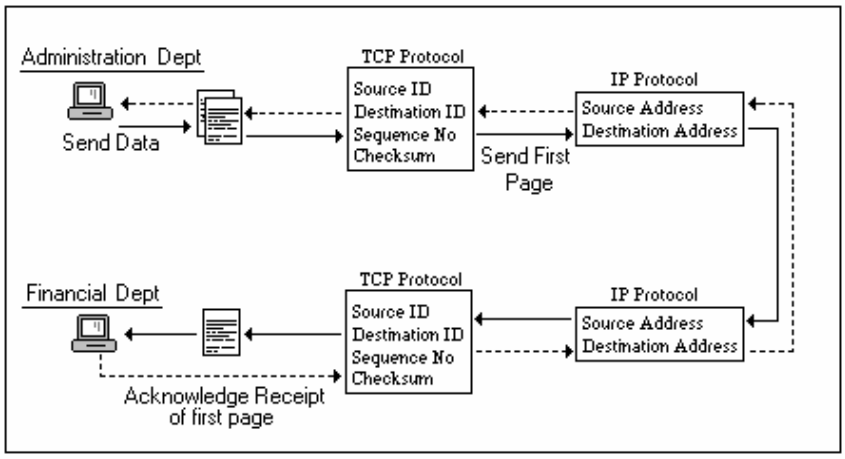
Before installing SOCKETS you need the following information regarding the network:

- Name and/or IP address of your workstation
- Type of Network Interface Controller (NIC) board on your PC
- I/O Port, IRQ and Memory Base Address settings of the controller
- Drivers that you intend to use, for example, packet driver, ODI driver or NDIS driver

TCP/IP Example

The following diagram provides an example of data being sent from an Administration department to a Finance department within a small company. The following steps are carried out:

1. Two pages of data are required to be sent, but the TCP protocol divides this into two chunks or segments.
2. TCP includes its associated header information then forwards the data to the IP protocol.
3. IP includes the relevant IP source and destination host addresses and forwards the data.



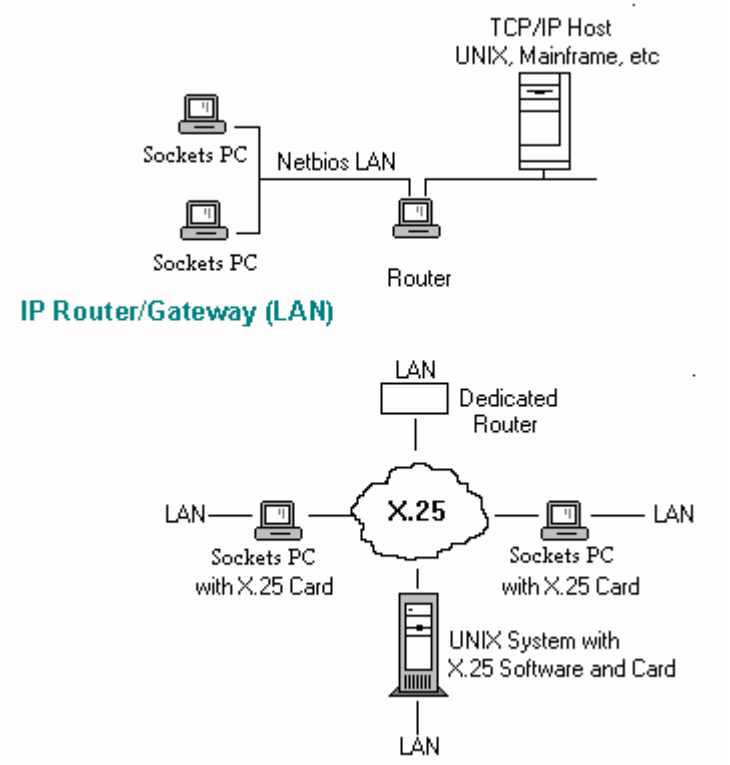
4. The data is received at the destination address and reconstructed by the TCP protocol.
5. The Financial department receives page 1 and sends an acknowledgement to the source address.
6. Page two of the data is sent through the same procedure.

This would be the situation on a one-system network; if the data were to be sent to another system on a different network, then the data would go through a routing procedure.

Client/Server Model

The most commonly used structure in distributed applications is the client/server model. In this model client applications request services from a server application. The client and server require a set of rules or protocols that must be implemented at both ends of the connection. The various protocols may act in a master/slave role, such as Telnet used for remote login, or may act in an equally responsive role such as the file transfer protocol (FTP).

Routing



Routing Information Protocol (RIP)

RIP allows routers to advertise available routes and endpoint workstations and allows routers to make use of the advertised routes to determine automatically the best path to a destination. Each RIP route has a metric or cost associated with it, as well as a limited lifetime, so that the network can dynamically adjust to route changes, such as the failure of a link or router in the network.

Routers or gateways are IP nodes on a network that connect more than one network together. This allows a workstation on one network to connect and communicate with a workstation on another network. Routers are often computers configured for more than one network interface. For example

when a segment is sent, if the destination network ID matches the source network ID then it is sent directly to the IP address. If the IDs do not match, the segment is sent to a router that knows the IDs of the other connected networks. The segment is forwarded to that router and then ultimately to the IP address. Routing can be applied to Wide Area Networks as well as Local Area Networks.

Address Resolution Protocol (ARP)

ARP provides a mechanism for hosts to determine the media access (MAC) addresses of Ethernet and Token Ring cards and map it to IP addresses automatically.

Simple Network Management Protocol (SNMP)

SNMP is a protocol providing an automated tool used by administrators to monitor and control TCP/IP based networks. It allows management workstations to monitor the state of TCP/IP hosts on the network and administrators to change networking parameters anywhere in the network. SNMP operates over UDP. SOCKETS has SNMP hooks available but does not ship with an SNMP agent.

SOCKETS (TCP/IP) Printing

SOCKETS printing is a method of using TCP/IP to perform network printing. Network printing means printing from any host on the network to a printer attached to any other host. The source of the print job uses a Print Client to open a TCP connection to a Print Server running on the host to which the destination printer is attached. The print data is sent over this connection from the Client to the Server, which passes the print data to the printer. The Client closing the TCP connection signals the end of the print job. Printer status information may be passed back to Client to signal error conditions such as “Paper out” or “Printer not ready.”

SOCKETS Operation Overview

The timer interrupt (INT 1CH) and the optional hardware interrupt(s) specified in the 'iface pdr ... h/w_int' command(s), if given, are hooked. Whenever one of these interrupts occurs or when the API is called, the SOCKETS 'scheduler' is called. The scheduler looks for queued incoming packets (queued at interrupt time by the Packet Driver) as well as the timer queue for timeouts, such as not receiving a TCP acknowledgment in time.

Packets are generally sent when the API is invoked, unless an ARP resolution is in progress, the TCP connection is not yet established, the Nagle heuristic is in operation or the offered window does not allow it. UDP packets are generally sent immediately when received by the API unless an ARP resolution is in progress.

Currently SOCKETM does *not* schedule on COM port interrupts.

Whenever SOCKETS is executing, it sets a BUSY flag and when the API is called during this time, which is only possible if it is called from an interrupt service routine, the API call fails with ERR_RE_ENTRY. Testing the BUSY flag before executing an API call can circumvent this error. The address of the BUSY flag can be obtained by the GET_BUSY_FLAG low-level API call.

Comment on the Timer Interrupt:

The timer is assumed to be the standard PC timer interrupt with a period of approximately 55 ms. Embedded systems using different timer periods should divide it down and provide a clock at INT 1CH, which is close to 55 ms (approximately 18 ticks per second - the exact value is 65536 ticks per hour).

Traffic Handling of SOCKETS:

This is very dependent on CPU speed. TCP throughput of more than 500 KB/s has been obtained using an AMD SC400 33Mhz and 10MB/s Ethernet

Chapter 5, SOCKETS Configuration and Use (Network & Transport Layers)

This chapter describes how to configure SOCKETS for use in several common environments.

Configuring SOCKETS

There are multiple levels of configuration available for SOCKETS. A series of command line options are available which are detailed within the Kernel Applications section of this chapter. These options configure the SOCKETS kernel itself. Other configurations happen while defining the interface. Examples are provided within the Configuration File Overview section of this chapter.

SOCKETS can be used as an IP gateway and/or a print server running concurrently with DOS or Windows applications. Since the presence of these applications necessarily changes the system, you should test the final environment carefully to ensure that there are no harmful interactions.

SOCKETS uses two files in the \SOCKETS directory (default), or any other directory specified by the SOCKETS environment variable. These files are SOCKET.CFG, the default start-up file, and HOSTS, the host names file. If not found, SOCKETS uses the default SOCKET.CFG in the SOCKETS directory. The SOCKETS environment variable is required in the demonstration version to indicate the directory with the LICENSE.DAT file, which contains the license information.

At the interface level, SOCKETS supports the packet driver interfaces. The following packet-driver classes are supported:

- Class 1 DIX Ethernet
- Class 3 Token Ring
- Class 6 SLIP
- Class 11 IEEE Ethernet

Before loading SOCKETS, make sure a packet driver is already in already loaded in memory. Most Ethernet packet drivers support both classes 1 and 11. As the default, Class 1 should normally be used.

Loading SOCKETS

If you want to use SOCKETS within a DOS prompt window, you must also load WINPKT before starting Windows.

Unloading SOCKETS

Run SOCKETP or SOCKETM with the /u switch to unload it from memory. SOCKETS disables itself if another program takes over its interrupts.

Kernel Applications

The Kernel Applications consist of the two executables SOCKETP and SOCKETM as well as the configuration file generator SCONFIG.

SOCKETP

SOCKETP is the SOCKETS/DOS TCP/IP kernel for use with packet drivers. It does not support direct access to serial devices. Run **SOCKETM** instead to enable support for serial devices.

Syntax

Socketp [/options] [config_file]

Options

/n=normal_sockets

/i=capi_interrupt

/d=dos_compatible_sockets

/m=memory_size

/p=print_delay

/s=stack_size

/v=interrupt_vector

/q

/0

/u

config_file [arguments_for_ config_file]

Example

config_file

If *config_file* is omitted, SOCKETS searches the following paths:
%SOCKETS%\DOS\SOCKET.CFG
%SOCKETS%\SOCKET.CFG
SOCKET.CFG

If the SOCKETS environment variable is not set, SOCKETS searches the following paths:
\DL\SOCKETS\DOS\SOCKET.CFG
\DL\SOCKETS\SOCKET.CFG
SOCKET.CFG

arguments_for_ config_file

Can be used as %1 to %9 in the configuration file. It is often used to set the IP address or other variable parameters in the start-up file. It can also be used to simplify many functions.

print_delay

If the printing speed is not satisfactory, print_delay is a number specifying the number of times a printer is tested for busy status before the busy status is accepted. This solves the problem that a single character is sent to the printer every timer tick (55 milliseconds), resulting in extremely slow

printing by the print server. print_delay should be the lowest value still permitting fast printing. The default value is 20.

normal_sockets
The number of normal sockets to reserve for use by the Compatibility API. If that API is unneeded, set this to 0. This value defaults to 8.

dos_compatible_sockets
The number of DOS compatible sockets to reserve for use by the Compatibility API. If that API is unneeded, set this to 0. This value defaults to 12.

capi_interrupt
The compatibility API uses interrupt 61h. To change it to a different value, specify capi_interrupt in hexadecimal.

sockets_interrupt
The SOCKETS API normally uses interrupt 7Fh.

memory_size
Specifies the working memory for SOCKETS and has a default value of 16384 (16KB).

stack_size
Specifies the stack size used by SOCKETS and has a default value of 2048 (2k).

/0
Instructs SOCKETS to not attempt to save and restore the extended register set of an 80386 processor. This is the only 386-specific code in SOCKETS/DOS, and disabling it makes SOCKETS/DOS fully 8086 compatible.

/q
Load SOCKETS without displaying any diagnostics.

/u
Run SOCKETS with the /u switch to unload it from memory. SOCKETS disables itself if another program takes over its interrupts.

SOCKETM

SOCKETM is the SOCKETS TCP/IP kernel with additional support for direct use of serial devices. In all other respects, it operates identically to **SOCKETP**.

Syntax

SOCKETM [/options] [config_file]

Remarks

Please see the options and example section for SOCKETP for specific reference.

SCONFIG

SCONFIG is a SOCKETS configuration utility. It supports only the barest configurations: Ethernet over a packet driver and PPP over a serial modem. Just answer the multiple choice questions, and then fill in a few data fields and **SCONFIG** writes simple *SOCKET.CFG* and *MODEM.MCF* files for you.

Syntax

SCONFIG

Configuration Considerations

MTU (Maximum Transmission Unit)

The MTU is the maximum size in bytes of a data packet (including all IP and TCP header information.) Information is sent across the Internet in packets, which are reassembled into a whole when they reach their destination. The size of these packets is dependent on the MTU of the machines along the route the packets travel.

The MTU can be specified by each individual computer. When you send out a request for information, the computer you are are requesting information from will read your request and hopefully send out packets of the size that you request. If the destination machine has a smaller MTU, the packets will be broken into pieces, or fragmented.

MSS (Maximum Segment Size)

MSS is the maximum size in bytes of the data portion of a TCP/IP packet This value is normally the MTU setting minus 40. MSS is used by SOCKETS to determine the WriteSocket queue size. The SOCKETS queuing overhead is 12 bytes per WriteSocket request so you will need 52 bytes for each request. SOCKETS will queue 2*MSS bytes if it can't send right away because of window constraints by the peer or the Nagle Algorithm. This means that if your MSS is set to the ethernet default of 1460, 2920 bytes will be queued, consuming 3796 bytes with overhead per connection. If you have five connections, you need 18980 bytes just to buffer your outgoing data. Use IPSTAT to determine how much memory you have available. (You need memory for each connection, incoming data, and all other features of the TCP/IP stack).

Buffers

Buffers are used to store data packets that are sent and received. The size of each buffer is determined by the MTU setting. Additionally, SOCKETS adds an additional 12 bytes for each WriteSocket request. The maximum amount of memory required for buffers is determined by the following formula

Maximum Buffer Memory Use = MTU * Buffers

A buffer limit of 30 is excessive if you have 1500 byte buffers (1500 * 30 = 45000). By default, SOCKETS allocates 20000 bytes, which is used for a lot of other processes as well, so you are bound

to encounter out-of-memory situations. If you set MTU to 1500, then 5 would be a recommended setting for the number of buffers.

Configuration Examples

The following examples within this chapter refer to specific files, such as "EXAMPLE1.CFG" and "MODEM.MC2". These example files can be found in the \SOCKETS\CONFIGS directory created during the install process.

- Ethernet connection with SOCKETS acting as a server to a Win9X or NT system. The DL web server can be launched and a standard desktop browser will "surf" to the SOCKETS system.
- Dial-Up serial connection to an ISP.
- Single Dial-In Connection.
- Single Dial-In connection with ASY interface.
- Direct serial connection with SOCKETS as a server.
- Direct serial connection with SOCKETS as a client.
- Dial-up slip connection with SOCKETS as an IP router.
- Multiple dial–in connections.

Example 1: Ethernet Connection – SOCKETS Serving a Web Page

SOCKETP is loaded from the command line as follows:

SOCKETP.EXE EXAMPLE1.CFG

To demonstrate the web interface, next launch HTTPD.EXE. If the index file is not within the current directory please remember to set the environment variable HTTP_DIR to the location of the index files or nothing will be displayed to the desktop browser.

```
SET HTTP_DIR=C:\DL\SOCKETS\SERVER

HTTPD.EXE /R
```

The configuration file EXAMPLE1.CFG contains:

Ip address 196.6.1.111/24	Sets an IP address of 196.6.1.111 with a 24-bit subnet mask to the following interface.
Iface pdr if0 dix 1500 5 0x60	Creates a standard Ethernet connection, type pdr (packet driver), interface name if0, type dix, MTU 1500, Buffer Limit 10, Interrupt vector of the packet driver at 0x60.
route add default if0 196.0.0.1	Set the default route for packets travelling to and from if0 to 196.0.0.1
domain server 196.0.0.1	Set the domain name server to 196.0.0.1 This server is used to convert names to IP addresses.
ip address	Cause SOCKETS to display IP status

# set TCP parameters	
tcp window 2920	Set the windows size to 2920 bytes
tcp retry 6	Set the retry count to 6
tcp irtt 500ms	Set the Initial Round Trip Time to 500ms
tcp mss 1460	Set the Maximum Segment Size to 1460 bytes

Example 2: Serial Connection – SOCKETS Dial-up to ISP

SOCKETM is loaded from the command line as follows:

SOCKETM.EXE EXAMPLE2.CFG

The configuration file EXAMPLE2.CFG contains:

interface asy p0 ppp 576 10 0x3f8 4 19200 modem.mc2	Creates an asynchronous ppp connection on COM1 IRQ4 named p0 with an MTU of 576 and buffer limit of 10. Uses the file modem.mc2 for modem configuration information.
route add default p0	Route all packets through interface p0
ip ttl 64	Set time to live for packets at 64 “hops”
tcp mss 1460	Set Maximum Segment Size to 1460 bytes
tcp window 2920	Set Windows Size to 2920 bytes
par p0 ipcp local compress tcp 16 1	Enables header compression, where 16 are the maximum number of concurrent TCP/IP connections. 1 turns on compression, 0 turns off compression.
par p0 ipcp local address 0.0.0.0	Server assigned IP address
par p0 lcp local accm 0	Asynch control character map set all bits to zero.
par p0 lcp local acfc on	Address control field compression on or off.
par p0 lcp local pfc on	Protocol field compression on or off.
par p0 lcp local magic on	Magic number option on or off. The magic number is used to detect loop back links by creating a number, sending a configure request, then comparing the number received. If it is the same then there is a possible loop back, repeat test.
par p0 pap user test example	Set the user name to “test” and password to “example”
par p0 ipcp open par p0 lcp open	Open specified layer. ipcp = ip control protocol. Lcp = link control protocol. When an immediate dial operation is required the "par if0 lcp start" command should be used. When dial-on-demand is desired the “par if0 lcp open" command should be used and when connect-on-

	received-call is desired, the "par if0 lcp listen" command should be used. The start command implies the open command and the open and listen commands may both be used. Dial-on-demand can also be disabled by the "p n" modem configuration file parameter and by an API call. Dial-on-demand can be enabled by an API call.

MODEM.MC2

i A@w100atdt^M^J@w100	Initialise modem
d ^M^J@w2000ATDT@n^M^J@d40000	Dialling string
n 123-4567	Number to dial
r 3	Number of retries
p rx	Enable debugging output to screen

Example 3: Single Dial-in Connection

Allow only single users to dial in and issue each with an IP address on the fly. Setup **asy** interfaces, each with its own IP address, COM port, modem and telephone number. In the answer script, a password may be asked (no need for XID: does it in the script), and on success an IP address is sent in dotted decimal form. (Should you want to use different passwords, an **mdd/aslink** setup as described in Example 2 may be used, but then every user will have his own IP address fixed to the XID.)

PPP Parameters:

Ipcp = ip control protocol.
Lcp = link control protocol.

The commands **lcpin, papin, apin** and **ipcpin** can be specified for incoming parameters. These correspond with the **lcp, pap, ap** and **ipcp** commands. The reason for having a different set of PPP parameters is to allow the SOCKETS implementation to act as both a “PPP server” and a “PPP client” without having to re-configure it.

The configuration file EXAMPLE3.CFG contains interface configurations for all modem ports as well as other connections (like Ethernet) that might be used. An **asy** interface could be:

IPAddress=192.6.3.1/30	Set the IP Address of the following interface to 192.6.3.1 with a subnet mask set to 30 bits.
iface asy sl0 cslip 576 10 0x3F8 4 19200 modem.mc3	An asy modem connection on COM1 connecting to the smallest subnet allowing 2 hosts.

The modem definition file is MODEM.MC3:

i a@w100atz^M^J@w500ats0=1^M^J@w100	Initialise modem (consult the manual for your own modem initialisation string)
x 192.6.3.2	Specify IP address (in the XID variable to make the answer scripts more uniform).
a @w1000^M^JPassword? @r9000Sockets@@w200^M^JYour address: @x ^M^J@r1000GOTIP@	Answer script to prompt remote for logon: Wait 1 second, send "<CR><LF>" to get cursor on a new line, send "Password? " and wait 9 seconds to receive "Sockets" somewhere in the data stream # (the connection will break if failed), wait 200 milliseconds and send the IP address with @x on a new line. Wait 9 seconds to receive "GOTIP" as confirmation.
p rx	Parameters for debugging (show modem data): r=receive and/or x=xmit

Example 4: Single Dial-in Connection with ASY Interface

A user who dials into the system in Example 3 which supplies the IP address for the user, has to use an **asy** interface to his modem. The user must specify the **@a** address command in the connect script in the modem file. The **@a *time*** command waits for ***time*** milliseconds to see the dotted decimal form of an IP address. It assigns this address to the **asy** interface. The user should have a default route to this **asy** interface to be able to see the whole world through it, or just a specific route for what he wants to see. The configuration file SOCKET4.CFG should contain at least the **asy** interface configuration similar to the previous examples.

PPP Parameters:

Ipcp = ip control protocol.
Lcp = link control protocol.

The commands **lcpin, papin, apin** and **ipcpin** can be specified for incoming parameters. These correspond with the **lcp, pap, ap** and **ipcp** commands. The reason for having a different set of PPP parameters is to allow the SOCKETS implementation to act as both a “PPP server” and a “PPP client” without having to re-configure it.

The modem definition file is MODEM.MC4:

i a@w100atz^M^J@w500ats0=0^M^J@w100	Initialise the modem (consult the manual for your own modem initialisation string).
d ^M^J@w2000atdt@n^M^J@d40000	Dial command
n 0800123456	Number to dial
r 5	# Number of retries
c @r7000sword@Sockets^M^J@a1000GOTIP	Connect script to login at the remote: wait 7 seconds after DCD to receive password prompt, send "Sockets" as password, wait 1

	second for IP address, send "GOTIP" as confirmation.
p rx	Parameters for debugging (show modem data): R=receive and/or x=xmit

Example 5: Direct Serial Connection with SOCKETS as a Server

This section explains how to configure SOCKETS as a server to listen on the designated serial port and wait for a valid connection and use standard dial up networking to connect.

The Client Connection

There are two methods of connection available - direct serial connection or dial in on a standard phone line. The Client here can be any Win95, Win98, or WinNT system.

For a direct serial connection a modem driver must be installed. Such a driver does not ship standard with SOCKETS but can be obtained as freeware at **http://www.aeriden.com**. The client must then create a dial up networking connection with the phone number of CLIEN1, no username or password. Please read the manual distributed with the aeriden driver to ensure correct setup.

Configuring SOCKETS

Configuration of the SOCKETS software is handled within two files, namely EXAMPLE5.CFG and MODEM.MC5. SOCKETS can be loaded by means of a batch file that contains the following commands:

SOCKETM EXAMPLE5.CFG

These commands allow SOCKETS to allocate more memory, than allowed by the default values, when initiating a TCP/IP connection over a serial link. SOCKETS processes the EXAMPLE5.CFG file followed by the MODEM.MC5 file.

SOCKETS Configuration File Details

PPP Parameters:

Ipcp = ip control protocol.
Lcp = link control protocol.

The commands **lcpin, papin, apin** and **ipcpin** can be specified for incoming parameters. These correspond with the **lcp, pap, ap** and **ipcp** commands. The reason for having a different set of PPP

parameters is to allow the SOCKETS implementation to act as both a “PPP server” and a “PPP client” without having to re-configure it.

EXAMPLE5.CFG contains:

iface asy if0 ppp 576 5 0x2f8 3 19200 modem.mc5	Creates an asynchronous connection on com 2, IRQ 3 with a Baud rate of 19200, MTU of 576, and a Buffer Limit of 5. Access the modem.mc5 file for specific initialization strings.
User SocketsUser SocketsPassword 196.10.229.18	For user connecting with name “SocketsUser” and Password “SocketsPassword” assign the ip address of 196.10.229.18
par if0 ipcpin local compress tcp 16 1	Enables header compression, where 16 are the maximum number of concurrent TCP/IP connections. 1 turns on compression, 0 turns off compression.
par if0 ipcpin local address 196.10.229.2	
par if0 ipcp local compress tcp 16 1	Enables header compression, where 16 are the maximum number of concurrent TCP/IP connections. 1 turns on compression, 0 turns off compression.
par if0 ipcp local address 196.10.229.4	
par if0 lcpin local accm 0	Asynch control character map, set all bits to zero.
par if0 lcpin local acfc on	Address control field compression on or off.
par if0 lcpin local pfc on	Protocol field compression on or off.
par if0 lcpin local magic on	Magic number option on or off. The magic number is used to detect loop back links by creating a number, sending a configure request, then comparing the number received. If it is the same, and then there is a possible loop back, repeat test.
par if0 lcpin local authen pap	Set local authentication to pap
par if0 lcp local accm 0	Asynch control character map set all bits to zero.
par if0 lcp local acfc on	Address control field compression on or off.
par if0 lcp local pfc on	Protocol field compression on or off.
par if0 lcp local magic on	Magic number option on or off. The magic number is used to detect loop back links by creating a number, sending a configure request, then comparing the number received. If it is the same, then there is a possible loop back, repeat test.
Par if0 ipcp open Par if0 ipcpin open par if0 lcp listen	Open specified layer. ipcp = ip control protocol. Lcp = link control protocol.

par if0 lcpin listen	When an immediate dial operation is required the "par if0 lcp start" command should be used. When dial-on-demand is desired the “par if0 lcp open" command should be used and when connect-on-received-call is desired, the "par if0 lcp listen" command should be used. The start command implies the open command and the open and listen commands may both be used. Dial-on-demand can also be disabled by the "p n" modem configuration file parameter and by an API call. Dial-on-demand can be enabled by an API call.
route add default if0	Route all packets through interface if0
ip address route	Prints out the systems ip address and route. Useful for debugging only.

MODEM.MC5

	Initialize Modem. For a direct serial connection no initialization string is necessary.
a @r20000CLIEN1@CLIENTSERVER	When Sockets receives the string “Clien1” send the response “ClientServer”
r 3	Number of Retries
p rx	Debugging information parameters: r=show modem receive, x=show modem xmit d=always dial, n=no dial-on-demand

Example 6: Direct Serial Connection with SOCKETS as a Client

Setting the Server

The NT server must have Remote Access Services enabled, commonly referred to as RAS, as well as the "Direct Serial cable between two PCs " modem driver installed on the proper COM port. The following example also connects using DHCP. If this feature is to be used then DHCP services must also be setup on the NT server. Other options would include the use of static IP addressing or BOOTP. Please review the cabling requirements from the WinNT documentation or help files. Standard cables often will not work with direct connections due to the cabling demands of WinNT.

Configuring the Software

Configuration of the SOCKETS software is handled within two files, namely EXAMPLE6.CFG and MODEM.MC4. SOCKETS can be loaded by means of a batch file that contains the following commands:

SOCKETM EXAMPLE6.CFG

These commands allow SOCKETS to allocate more memory than allowed by the default values when initiating a TCP/IP connection over a serial link. SOCKETS processes the ECAMPLE6.CFG file followed by the MODEM.MC6 file.

Configuration File Details

PPP Parameters:

Ipcp = ip control protocol.
Lcp = link control protocol.

The commands **lcpin, papin, apin** and **ipcpin** can be specified for incoming parameters. These correspond with the **lcp, pap, ap** and **ipcp** commands. The reason for having a different set of PPP parameters is to allow the SOCKETS implementation to act as both a "PPP server" and a "PPP client" without having to re-configure it.

EXAMPLE6.CFG:

Iface asy if0 ppp 1500 30 0x2f8 3 19200 modem.mc6	Creates an asynchronous connection on com 2, IRQ 3 with a baud rate of 19200 and access the modem.cfg file for specific initialization strings.
par if0 ipcp local compress tcp 16 1	Enables header compression, where 16 are the maximum number of concurrent TCP/IP connections. 1 turns on compression, 0 turns off compression.
par if0 ipcp local address 0.0.0.0	The ip address 0.0.0.0 indicates that the server must assign the client a valid ip address
par if0 lcp local accm 0	Asynch control character map, set all bits to zero.
par if0 lcp local acfc on	Address control field compression on or off.
par if0 lcp local pfc on	Protocol field compression on or off.
par if0 lcp local magic on	Magic number option on or off. The magic number is used to detect loop back links by creating a number, sending a configure request, then comparing the number received. If it is the same, then there is a possible loop back, repeat test
par if0 pap user id password	Replace "id" and "password" with a valid username and password on the NT server that you are connecting to.
par if0 ipcp open par if0 lcp open	Open specified layer. ipcp = ip control protocol. lcp = link control protocol.
route add default if0	Route all packets to interface if0
ip address	Prints to the screen the systems ip address

route	and route, useful for debugging only.

MODEM.MC6

i CLIENT@r5000CLIENTSERVER@@a2000	Initialize Modem and connect as client
r 3 p rx	Number of Retries Debugging parameters r display what we receive x display what we transmit

Example 7: SOCKETS Machine Using Call Back Verification.

Scenario:

If the NT machine logs into the SOCKETS machine with SocketsUser1 / SocketsPassword1, the NT machine is assigned IP address 196.10.229.18 and the SOCKETS machine is 196.10.229.2.

If the NT machine logs into the SOCKETS machine with "SocketsUser / SocketsPassword," the SOCKETS machine will break the connection and dial back to 08036501 logging in as "NtUser" with password "NtPassword." The NT machine assigns the IP addresses of both machines.

The SOCKETS machine never initiates a modem call unless it has been called first. Once SOCKETM is running, the IOCTL program can be used to initiate a dial operation or to enable dial on demand.

SOCKETS Configuration File Details

PPP Parameters:

Ipcp = ip control protocol.
Lcp = link control protocol.

The commands **lcpin, papin, apin** and **ipcpin** can be specified for incoming parameters. These correspond with the **lcp, pap, ap** and **ipcp** commands. The reason for having a different set of PPP parameters is to allow the SOCKETS implementation to act as both a "PPP server" and a "PPP client" without having to re-configure it.

EXAMPLE7.CFG

iface asy p ppp 1500 30 0x3f8 4 9600 modem.mc7	Creates an asynchronous connection designated interface "p" type ppp on COM1 IRQ4 with an MTU of 1500, Buffer limit of 30, baud rate 9600 and modem information contained within the modem.mc7 file
route add default p	Routes all packets through interface "p"

user SocketsUser SocketsPassword 196.10.229.18 cbv	If user is SocketsUser with password SocketsPassword assign IP address of 196.10.229.18 then disconnect session and call back.
user SocketsUser1 SocketsPassword1 196.10.229.18	If user is SocketsUser1 with password SocketsPassword1 assign IP address of 196.10.229.18 and continue session.
par p ipcpin local compress tcp 16 1	Enables header compression, where 16 are the maximum number of concurrent TCP/IP connections. 1 turns on compression, 0 turns off compression.
par p ipcpin local address 196.10.229.2	Assign local IP address 196.10.229.2
par p ipcp local compress tcp 16 1	Enables header compression, where 16 are the maximum number of concurrent TCP/IP connections. 1 turns on compression, 0 turns off compression.
par p ipcp local address 0.0.0.0	The ip address 0.0.0.0 indicates that the server must assign the client a valid ip address.
par p lcpin local accm 0	Sets the Asynch Control Character Map to 0.
par p lcpin local acfc on	Sets local Address and Control Field Compression on.
par p lcpin local pfc on	Sets local Protocol Field Compression on.
par p lcpin local magic on	Magic number option on or off. The magic number is used to detect loop back links by creating a number, sending a configure request, then comparing the number received. If it is the same, then there is a possible loop back: repeat test
par p lcpin local authen pap	Sets the local Authentication protocol to pap.
par p lcp local accm 0	Sets the local Asynch Control Character Map to 0.
par p lcp local acfc on	Sets the local Address and Control Field Compression to on.
par p lcp local pfc on	Sets the local Protocol Field Compression to on.
par p lcp local magic on	Magic number option on or off. The magic number is used to detect loop back links by creating a number, sending a configure request, then comparing the number received. If it is the same, then there is a possible loop back, repeat test. .
par p pap user NtUser NtPassword	Allow login for user NtUser with password NtPassword. PAP corresponds with the previous set authentication protocol.
par p ipcp open	Set incoming dial-in method to open.
par p ipcpin open	Set incoming dial-in method to open.
par p lcp listen	Set dial-out method to listen.

par p lcpin listen	Set dial-out method to listen.

MODEM.MC7

i a@w500a@w500atl1w1%e2&q5&c1&d2s11=70&k3s30=18s0=0^M^J@w100	Modem Initialization string. Please refer to the modem manufacturer manual for the specific initialization string to match your modem.
a	
n 08036501	Number to dial
d ^M^J@w1000atd@w110@n@w110^M@d40000	Dial command
c	
r 4	Number of retries to dial the connection.
p n	Debugging parameters: r=show modem receive, x=show modem xmit d=always dial, n=no dial-on-demand

Example 8: SOCKETS Machine with CBV and Logging-in.

PPP Parameters:

Ipcp = ip control protocol.
Lcp = link control protocol.

The commands **lcpin, papin, apin** and **ipcpin** can be specified for incoming parameters. These correspond with the **lcp, pap, ap** and **ipcp** commands. The reason for having a different set of PPP parameters is to allow the SOCKETS implementation to act as both a "PPP server" and a "PPP client" without having to re-configure it.

EXAMPLE8.CFG

iface asy p ppp 1500 30 0x3f8 4 9600 modem.mc8	Creates an asynchronous connection designated interface "p" type ppp on COM1 IRQ4 with an MTU of 1500, Buffer limit of 30, baud rate 9600 and modem information contained within the modem.mc8 file
route add default p	Routes all packets through interface "p"
user NtUser NtPassword 196.10.229.208	If user is NtUser with password NtPassword assign IP address of 196.10.229.208 then disconnect session
par p ipcp local compress tcp 16 1	Enables header compression, where 16 are the maximum number of concurrent TCP/IP connections. 1 turns on compression, 0 turns off compression.
par p ipcp local address 0.0.0.0	The ip address 0.0.0.0 indicates that the

	server must assign the client a valid ip address.
par p lcp local accm 0	Sets the local Asynch Control Character Map to 0.
par p lcp local acfc on	Sets the local Address and Control Field Compression to on.
par p lcp local pfc on	Sets the local Protocol Field Compression to on.
par p lcp local magic on	Magic number option on or off. The magic number is used to detect loop back links by creating a number, sending a configure request, then comparing the number received. If it is the same, then there is a possible loop back, repeat test.
par p pap user SockectsUser SocketsPassword	Allow login for user SocketsUser with password SocketsPassword. PAP corresponds with the previous set authentication protocol.
par p ipcp open	Set incoming dial-in method to open.
par p lcp open	Set incoming dial-in method to open.
par p ipcpin local compress tcp 16 1	Enables incoming header compression, where 16 are the maximum number of concurrent TCP/IP connections. 1 turns on compression, 0 turns off compression.
par p ipcpin local address 196.10.229.202	Set the local IP address on an incoming transmission to 196.10.229.202
par p lcpin local accm 0	Sets the local Asynch Control Character Map to 0.
par p lcpin local acfc on	Sets the local Address and Control Field Compression to on.
par p lcpin local pfc on	Sets the local Protocol Field Compression to on.
par p lcpin local magic on	Magic number option on or off. The magic number is used to detect loop back links by creating a number, sending a configure request, then comparing the number received. If it is the same, then there is a possible loop back: repeat test
par p lcpin local authen pap	Set the authentication protocol for incoming lcpin to pap.
par p ipcpin open	Set dial-in method for ipcpin to open.
par p lcpin open	Set dial-in method for lcpin to open

MODEM.MC8:

I	Initialize the modem; please refer to the

a@w500a@w500atl1w1%e2&q5&c1&d2s11 =70&k3s30=18s0=0^M^J@w100	manufacturer documentation for the specific modem initialization string.
a	
n 08034131	Number to dial
d ^M^J@w1000atd@w110@n@w110^M@d4 0000	Dial string.
c	
r 4	Number of retries.
p r	Debugging information. Parameters: r=show modem receive, x=show modem xmit d=always dial, n=no dial-on-demand

When the "NT look-alike" side starts up, it dials 08034131. The "SOCKETS" side answers and after the successful login, drops the connection and dials 08036501. During PPP negotiation, the "NT look-alike" side is assigned IP address 196.10.229.202 and the "SOCKETS" side, IP address 196.10.229.108.

Example 9: Dial-up SLIP Connection with SOCKETS as an IP Router

You want to connect your LAN to another network via a modem using a dial-up SLIP link. The SOCKETS PC will act as an IP router (or gateway) to the other network. This example is symmetrical: You can use the same setup on both sides to enable any side to initiate the call. (The IP addresses will have to be unique.)

Use an **asy** interface with CSLIP to get better throughput. Your network is 192.6.1.0 and your address 192.6.1.111. The **asy** interface will also be 192.6.1.111 linking the 192.6.2.0 network (or the rest of the world) on the other side. The modems establish a connection with no logon required.

PPP Parameters:

Ipcp = ip control protocol.
Lcp = link control protocol.

The commands **lcpin, papin, apin** and **ipcpin** can be specified for incoming parameters. These correspond with the **lcp, pap, ap** and **ipcp** commands. The reason for having a different set of PPP parameters is to allow the SOCKETS implementation to act as both a "PPP server" and a "PPP client" without having to re-configure it.

The configuration file EXAMPLE9.CFG contains:

Ip address 196.6.1.111/24	Sets an IP address of 196.6.1.111 with a 24-bit subnet mask to the following interface.
Iface pdr if0 dix 1500 10 0x60	Creates a standard Ethernet connection, type pdr (packet driver), interface name if0, type dix, MTU 1500, Buffer Limit 10, Interrupt vector of the packet driver at 0x60.
IPAddress=192.6.1.111/0	Sets an IP address of 192.6.1.111 with a 0-bit

	subnet mask to the following interface.
Iface asy sl0 cslip 576 10 3f8 4 19200 asy-io.mod	Creates a standard modem connection on COM1. The connection is asynchronous, interface name sl0, MTU 576, Buffer Limit 4, baud rate 19200, and reference the file asy-io.mod for specific modem information.
; Make this the default route:	
par sl0 19200 CTS RTS	For interface sl0 set the serial baud rate to 19200, output flow to CTS, input flow to RTS. CTS = Clear to Send modem signal. RTS = Request To Send modem signal. *Use only for asy type interfaces.
par sl0 ripadv	Enable Routing Information Protocol on interface sl0.
rip advertise 30	Set RIP advertise time to 30. Time is the number of elapsed seconds before the advertisement is repeated.
rip use 200	Set RIP use time to 200. Time is the period during which routes are added or amended as a result of RIP, and are valid for time seconds.

The start up file SOCKETS.STU will display the current status with:

ip address	Display the IP address
route	Display the routes
ifstat	Display the interfaces

The modem definition file is ASY-IO.MOD:

I a@w100atz^M^J@w500ats0=1^M^J@w100	Initialise modem (Caution: consult the manual for your own modem)
d ^M^J@w2000atdt@n^M^J@d40000	Dial command: send "<CR><LF>", wait 2 seconds, send "atdt<number><CR><LF>", wait 40 seconds for DCD
n 0,011-790-1234	The number to dial
r 5	Number of retries
p r	Parameters for debugging (show modem data): r=receive and/or x=xmit
	Password protected logon facilities can be added with answer and connect scripts. (See next examples or ASY-IOL.MOD on disk)

Example 10: Multiple Dial-in Connections

You have a number of modems for dial-in of various users or other networks each with a fixed IP address. When somebody dials into your system, SOCKETS will need to know his IP address to be able to set up a return route to him. The network size for each remote host must also be known. (You have to understand sub-networks for this example.)

The way to do this is to set up a **mdd** interface for each IP address (using an IP address on that network). Each **mdd** interface has a modem definition file containing a unique exchange identifier (XID). Each modem has an **aslink** interface with a modem file containing an answer script. When the remote dials in, he has to specify his XID in his connect script (like a login). The **aslink** then links to the **mdd** with the same XID and your remote user is connected using the routes set for his **mdd** interface.

PPP Parameters:

Ipcp = ip control protocol.
Lcp = link control protocol.

The commands **lcpin, papin, apin** and **ipcpin** can be specified for incoming parameters. These correspond with the **lcp, pap, ap** and **ipcp** commands. The reason for having a different set of PPP parameters is to allow the SOCKETS implementation to act as both a "PPP server" and a "PPP client" without having to re-configure it.

The configuration file EXAMPL10.CFG should contain:

IPAddress=197.55.2.9	Set the IP address of the following interface to 197.55.2.9
iface pdr if0 dix 1500 10 0x60	Creates a standard Ethernet connection, type pdr (packet driver), interface name if0, type dix, MTU 1500, Buffer Limit 10, Interrupt vector of the packet driver at 0x60.
; A Multi Destination Driver	
; connecting to the smallest subnet allowing 2 hosts	
IPAddress=193.101.51.1/30	Set the IP address of the following interface to 193.101.51.1 with a 30-bit subnet mask.
iface mdd mdd0 slip 576 10 mdd10.mod	Create a multiple destination interface, type mdd, interface name mdd0, class slip, MTU 576, Buffer Limit 10, access the file mdd10.mod for specific interface commands.
iface asy as10 cslip 576 10 0x3F8 4 19200 aslink.mod	Create an aslink connection on COM1.

All the modems (**aslink** interfaces) may use the ASLINK.MOD file:

i	Initialise modem (Warning: consult the

a@w100atz^M^J@w500ats0=1^M^J@w10 0	manual for your specific modem initialisation string)
a @w500^M^JXID?@f5000@w200ccc	Answer script to prompt for XID
p r	Parameters for debugging (show modem data): r=receive and/or x=xmit

All the **mdd** interfaces will use a specific MDD*n*.MOD file with its own unique XID. The XID links to the IP address in the configuration of the **mdd** interface. MDD0.MOD will be:

x id0	Set the XID variable
p r	Parameters for debugging (show modem data): r=receive and/or x=xmit

The users dialling into the system above may use **asy** interfaces with a modem file containing the following commands (see file ASY-OX.MOD):

i a@w100atz^M^J@w500ats0=1^M^J@w100	Initialise modem (Warning: consult the manual for your own modem)
d ^M^J@w2000atdt@n^M^J@d40000	Dial command: send "<CR><LF>" wait 2 seconds, send "atdt<number><CR><LF>" wait 40 seconds for DCD.
n 790-1234	The number to dial
r 5	Number of retries
x id0	Set the XID variable
c @r1000XID@^F@x^M@r100c	Connect script to login at the remote: wait 1 second after DCD to receive XID prompt, send the ^F XID ^M sequence, receive a "c" as confirmation.
p rx	Parameters for debugging (show modem data): r=receive and/or x=xmit

Testing with XPING

The **XPING.EXE** (external ping) program gives a quick method to test your SOCKETS installation. The source code is also supplied as an example. XPING starts a continuous string of pings until stopped by a keystroke.

Syntax

XPING IP_address [interval]

Where the *IP_address* may be a numeric or symbolic address and the *interval* is the time to wait between pings in timer clock ticks. The default is 10 ticks.

Displaying Statistics with IPSTAT

The IPSTAT.EXE program gives statistics on IP and memory. Use it to check for error conditions and memory problems. It takes no arguments and a typical output is:

```
IP stats at 160F:04C8:
Total packets                2671
Smaller than minimum size       0
IP header length too small      0
Wrong IP version                0
IP header checksum errors       0
Unsupported protocol            0
Memory available             9016
Memory allocation failures      0
Memory free errors              0
Minimum stack observed        886
```

Use the TCP.EXE program described below for status and control of IP.

TCP Control and Status

The initial TCP parameters are set using the start-up file SOCKET.CFG or as specified on the command line. Use TCP.EXE to examine and change these parameters or to give a continuous updated display of the connection status. Run TCP without arguments to display help on the syntax.

Syntax

tcp close *n*	Close connection *n*
tcp irtt [*n*]	Set or display the Initial Round Trip Time
tcp mss [*n*]	Set or display the Maximum Segment Size
tcp reset *n*	Reset connection n
tcp retry [*n*]	Set or display the retry count
tcp status [/*columns* [*refresh_time*]]	Display summary status of all connections
tcp status n	Display detailed status for connection n
tcp window [n]	Set or display window size

Parameters

/*columns* is the number of columns in which to display status with manual or automatic refresh.

refresh_time is the time in seconds after which an automatic refresh occurs.

SOCKETS TCP Socket Print Services Configuration

SOCKETS includes both a socket print server and a printer redirector which can be used on up to four printer ports. The print server sends output to a printer via Interrupt 17. The printer redirector operates by intercepting Interrupt 17 and passing printer output to a local or remote print server. A print session is started when output is first sent to a specific port and stopped after a user-specified period of no output. To start printer services, you must use the start prntserv command in the example below. For a full reference to the "start" command please refer to Chapter 6.

When a printer port is redirected, the IP address and TCP port number of the destination print server, as well as the timeout period, is specified. A printer port can be used both as a print server port and as a redirected port. To ensure proper queuing of remote and local print sessions to a local printer; the printer port must be redirected to the local IP address and TCP port of the print server using that port.

To redirect a port use the printer command in SOCKET.CFG as follows:

Syntax

printer printer_port timeout IP_address [TCP_port]

Parameters

printer_port is printer port 0,1,2 or 3 to redirect. Port 0 corresponds with PRN, 1 with LPT2 and so on. You can link any port number to a serial port by using the SRPRINT TSR for serial printers.

timeout is the timeout period in seconds.

IP_address is the IP address of print server host.

TCP_port is the TCP port of print server. (Default is 10).

Example

```
printer 0 15 print_host
```

where *this_host* must be defined in HOSTS. To use printer port 1 (LPT2) for both local and remote printing:

```
ip address this_host
start prntserv 11 1
printer 1 15 this_host 11
```

Note: If the printer command is placed before start `prntserv`, SOCKETS will not function properly.

PPP Functionality

PPP Features

Support for "PPP server" or PPP dial-in and a Callback Verification facility is included. The raising of the Carrier Detect (DCD) signal signals an incoming call. Consequently, the modem must be configured to raise DCD only when the modem has answered the call. Log-in is controlled by a list of

Username/Password pairs, each coupled to a remote IP address, which is assigned to the peer if requested to do so during the IPCP negotiation.

As an option, a Username/Password pair can be flagged to provide Callback Verification (CBV). In this case, the call is terminated as soon as the PPP negotiations have been successfully completed. Then, a call is made to the number (or numbers) specified in the modem configuration file. During the first subsequent set of PPP negotiations, the CBV flag is ignored to prevent another callback. When the PPP session terminates, the CBV flag is again enabled.

Another PPP feature is the ability to specify that a PPP session should start immediately when SOCKETM is loaded, or to delay that until traffic is generated. Earlier versions of SOCKETS allowed only dial-on-demand.

It is also possible to specify two sets of PPP parameters per interface. The first set of parameters is for outgoing modem connections; the second set is for incoming modem connections.

Commands in the modem file can specify that a dial-up connection will always be dialed whenever DCD is not detected, or specify that dial on demand is disabled.

An API function, **IfaceIOCTL()**, controls asynchronous ports. **IfaceIOCTL()** allows a user program to:

- Initiate a modem connection (dial)
- Disconnect a modem connection
- Enable or disable dial on demand
- Enable or disable a serial port to allow a user program to access it without conflict
- Read modem and connection status.

A sample program, IOCTL.EXE, shows the use of this function and is provided in source and binary format in the SOCKETS Software Development Kit.

Username/Password List

Each Username/Password is specified by a **user** command in the SOCKETS configuration file in the format:

user *username password hostid* [cbv]

For example:

```
user myuser mypassword 196.10.229.2
user callbackuser callbackpassword 10.0.0.22 cbv
```

As many **user** commands as required may be used.

Immediate or Dial-on-demand Connections

When an immediate dial operation is required the **par** *ppp* **lcp open** command is used. When dial-on-demand or connect-on-received-call is desired, the **par** *ppp* **lcp listen** command is used. Dial-on-

demand can be disabled by the **p n** modem configuration file parameter and by a call to **IfaceIOCTL().** Conversely, dial-on-demand can be enabled by **IfaceIOCTL().**

Alternative Configuration Commands for Incoming Modem Connections

A set of PPP parameters to be used only for incoming modem connections must be specified. Four commands; **lcpin**, **papin, apin** and **ipcpin** can be specified In the **par** *ppp …* commands. These correspond with the **lcp**, **pap, ap,** and **ipcp** commands. Having a different set of PPP parameters allows the SOCKETS implementation to act as both a PPP server and a PPP client without having to be re-configured.

Modem Configuration File Additions

Two additional parameters can be used in the modem configuration file parameter command. They are *d* for always dialing when DCD is down and *n* for no dial-on-demand. These two parameters are mutually exclusive.

IfaceIOCTL Function

The IfaceIOCTL function controls asynchronous interfaces.

C syntax

int **IfaceIOCTL**(char *pszName, WORD wFunction);

Parameters

pszName

Pointer to interface name.

WFunction

Function to perform:

IOCTL_CONNECT	Start dial operation
IOCTL_DISCONNECT	Disconnect modem
IOCTL_ENABLEPORT	Enable communications port
IOCTL_DISABLEPORT	Disable communications port
IOCTL_ENABLEDOD	Enable dial-on-demand
IOCTL_DISABLEDOD	Disable dial-on-demand
IOCTL_GETSTATUS	Get modem/connection status

Return Value

Returns –1 on error, >= 0 if OK.

IOCTL_GETSTATUS returns the following bits:

ST_DTR	0x01	Data Terminal Ready
ST_RTS	0x02	Request To Send
ST_CTS	0x10	Clear To Send
ST_DSR	0x20	Data Set Ready
ST_RI	0x40	Ring Indicator
ST_DCD	0x80	Data Carrier Detect

ST_CONNECTED	0x100	Modem is connected
ST_MODEMSTATE	0xe00	Modem state mask
STM_NONE	0x000	No modem on port
STM_IDLE	0x200	Modem is idle
STM_INITIALIZING	0x400	Modem is initializing
STM_DIALING	0x600	Modem is dialing
STM_CONNECTING	0x800	Modem is connecting
STM_ANSWERING	0xa00	Modem is answering

Sample Configuration Files

Example 1: Configuration file of a SOCKETS machine being called by NT and calling back to the NT machine.

```
iface asy p ppp 576 5 0x3f8 4 9600 modem.mcf
route add default p
user SocketsUser SocketsPassword 196.10.229.18 cbv
user SocketsUser1 SocketsPassword1 196.10.229.18
par p ipcpin local compress tcp 16 1
par p ipcpin local address 196.10.229.2
par p ipcp local compress tcp 16 1
par p ipcp local address 0.0.0.0
par p lcpin local accm 0
par p lcpin local acfc on
par p lcpin local pfc on
par p lcpin local magic on
par p lcpin local authen pap
par p lcp local accm 0
par p lcp local acfc on
par p lcp local pfc on
par p lcp local magic on
par p pap user NtUser NtPassword
par p ipcp open
par p ipcpin open
par p lcp listen
par p lcpin listen

i a@w500a@w500atl1w1%e2&q5&c1&d2s11=70&k3s30=18s0=0^M^J@w100
a
n 08036501
d ^M^J@w1000atd@w110@n@w110^M@d40000
c
r 4
# parameters: r=show modem receive, x=show modem xmit
#             d=always dial, n=no dial-on-demand
p n
```

If the NT machine logs into the SOCKETS machine with “SocketsUser1 / SocketsPassword1,” the NT machine is assigned IP address 196.10.229.18 and the SOCKETS machine is 196.10.229.2.

If the NT machine logs into the SOCKETS machine with “SocketsUser / SocketsPassword,” the SOCKETS machine will break the connection and dial back to 08036501 logging in as “NtUser” with password “NtPassword.” The NT machine assigns the IP addresses of both machines.

The SOCKETS machine never initiates a modem call unless it has been called first. Once SOCKETM is running, the IOCTL program can be used to initiate a dial operation or to enable dial on demand.

Example 2: Configuration file for SOCKETS machine operating in the same manner as the NT machine in Example 1 above, and logging-in with callback username and password.

```
iface asy p ppp 576 5 0x3f8 4 9600 modem.mcf
route add default p
user NtUser NtPassword 196.10.229.208
par p ipcp local compress tcp 16 1
par p ipcp local address 0.0.0.0
par p lcp local accm 0
par p lcp local acfc on
par p lcp local pfc on
par p lcp local magic on
par p pap user SockectsUser SocketsPassword
par p ipcp open
par p lcp open
par p ipcpin local compress tcp 16 1
par p ipcpin local address 196.10.229.202
par p lcpin local accm 0
par p lcpin local acfc on
par p lcpin local pfc on
par p lcpin local magic on
par p lcpin local authen pap
par p ipcpin open
par p lcpin open

Modem configuration file:

i a@w500a@w500atl1w1%e2&q5&c1&d2s11=70&k3s30=18s0=0^M^J@w100
a
n 08034131
d ^M^J@w1000atd@w110@n@w110^M@d40000
c
r 4
# parameters: r=show modem receive, x=show modem xmit
#             d=always dial, n=no dial-on-demand
p r
```

When the “NT look-alike” side starts up, it dials 08034131. The “SOCKETS” side answers and after the successful login, drops the connection and dials 08036501. During PPP negotiation, the “NT look-alike” side is assigned IP address 196.10.229.202 and the “SOCKETS” side, IP address 196.10.229.108.

Using PPP with SOCKETS

Configuring the PPP Interface

The PPP interface is optionally configured by using the **par** command described on page 67. Various parameters at the LCP, IPCP and PAP levels can be specified.

To define an interface for PPP, use the following Interface command.

interface asy ifname **ppp** mtu buflim port int speed [modemfile]

Example

```
iface asy p0 ppp 576 5 0x3f8 4 9600 pppmod.mod
```

Parameters

ifname defines the name used in the interface command for the device to be controlled.

mtu specifies the Maximum Transmission Unit size in bytes. Datagrams larger than *mtu* are fragmented into smaller pieces at the IP layer. A typical value for *mtu* is 1500. For serial links a standard value for *mtu* is 576. (576 are the maximum according to specifications, but may be increased on reliable connections as long as both sides use the same value.)

buflim specifies the maximum number of outgoing datagrams or packets to queue before starting to discard datagrams. This mechanism prevents memory from filling when a serial link goes down.

port specifies the port address of the COM port.

int specifies the IRQ number for the COM port.

speed specifies the transmission speed for serial interface devices (baud rate). Before using a serial connection, set flow control with the **par** command.

modemfile is a file containing the modem commands and scripts.

Setting PPP Options, Local and Remote LCP/IPCP

When a local option is specified, the value of the option is used in the initial Configuration Request to the peer. Options not specified, are not be requested. For each option, the 'allow off' parameter disallows the peer to include that option in its response. By default, all options are allowed in the response, even if the option is not included in the request.

When a remote option is specified, the value of the option is used in the initial response to the configuration request from the peer. If an option is disallowed, it does not allow the remote to specify that option in its request. By default all options are allowed.

Local and remote options are specified by:

par *iface* lcp|ipcp local|remote *option* [*parameters* ...] [allow [on|off]]

The **par** command options are as follows:

Syntax	Description
par *iface* lcp local|remote accm *bitmap*	Set the Asynch Control Character Map. The default is 0xffffffff.
par *iface* lcp local|remote authent [pap|chap|none|allow [on|off]]	Set the Authentication protocol. The default is none.
par *iface* lcp local|remote acfc [on|off|allow [on|off]]	Set Address and Control Field Compression. The default is off.
par *iface* lcp local|remote pfc [on|off|allow [on|off]]	Set Protocol Field Compression. The default is off.
par *iface* lcp local|remote magic [on|off|*value*|allow [on|off]]	Set the Magic number option (detects looped back circuits)
par *iface* lcp local|remote mru [*size*|allow [on|off]]	Set the Maximum Receive Unit. The default is 1500.

Syntax	Description
par *iface* lcp [open|close]	Set the Sockets dial-out method. The default is close, preventing dial-out until a command has been issued to the stack.
par *iface* ipcp local|remote address [*ip_address*>|allow [on|off]]	Set the IP address option. If set to 0.0.0.0, the peer must supply it.
par *iface* ipcp local|remote compress tcp *slots* [*flag*] allow [on|off]]	Set the TCP/IP header compression. *slots* should be equal to the maximum number of concurrent TCP connections. Low values preserve memory. 4 - 16 are good values. *flag* = 0 to not compress the slot number; and =1 to compress. The default is 1.

Setting PPP Options, Retry Counters

The following retry counters can be set:

par <iface> lcp retry <configure>|<failure>|<terminate> <count>

par <iface> ipcp retry <configure>|<failure>|<terminate> <count>

iface is the name assigned to the interface.

configure is the number of configuration requests (default 20).

failure is the number of bad configuration requests allowed from peer (10).

terminate is the number of termination requests before shutdown (2).

count is the number of retries.

Setting PPP Options, Timeout Values - in milliseconds

par *iface* lcp|ipcp|pap timeout *milliseconds*

Setting PPP Options, Authentication - username/password

For PAP authentication on PPP connections:

par iface pap user username password

For CHAP authentication on PPP connections

par iface ap user username password

Setting PPP Options, Open - a specified layer

par *iface* lcp|ipcp open

Sample Startup File

```
iface asy ppp0 ppp 576 5 0x3f8 4 9600 pppmod.mod
```

```
route add default ppp0
par ppp0 ipcp local compress tcp 16 1
par ppp0 ipcp local address 0.0.0.0
par ppp0 local accm 0
par ppp0 lcp local acfc on
par ppp0 lcp local pfc on
par ppp0 lcp local magic on
par ppp0 pap user tstppp xyz
par ppp0 ipcp open
par ppp0 lcp open
```

This example can be used to connect to an NT RAS server. In this example, a user with name **tstppp** and password **xyz** must be set up on NT and have dial-in permission. Also make sure that the Server settings reached with Control panel|Network|Services|Remote Access Service|Properties|Remote Access Setup|Network is set as follows:

- TCP/IP checked.
- Encryption settings: Allow any authentication including clear text.

It is necessary to start SOCKETM with more than the default amount of memory:

SOCKETM /m=20000 ppp It is necessary to start SOCKETM with more than the default amount of memory:

```
SOCKETM /m=20000 ppp
```

Chapter 6, SOCKETS Configuration Reference (Network &Transport Layers)

Overview

This chapter describes the SOCKETS commands available when starting and running SOCKETS on the target system. To run any of the commands described in the following sections, insert the appropriate entries in the SOCKET.CFG file.

Notations and Conventions

In the command summary, use is made of the *hostid* notation, which denotes a host, router, gateway, or network. *hostid* may be specified either by a symbolic name listed in the HOSTS file, or a numeric IP address with decimal notation; for example, 192.10.240.1.

The following conventions apply to command syntax.

- *italics* indicate that the term is a parameter to be specified by the user.
- [ ] Square brackets indicate that the enclosed item is optional.
- / A forward slash is used as a leading character for an optional switch in some commands. Both the forward slash and the switch that follows it form part of the command syntax. The equal sign before extra parameters is optional in most cases.
- | The vertical bar indicates that there is a choice between two or more selections, but that only one of the options indicated may be specified.
- **bold** type indicates a reserved key word as part of the command syntax and is to be typed exactly as indicated.
- # Commands proceeded by a hash sign (#) are ignored. They are used for comments in start-up or command files.

Command Reference

arp

arp includes a new entry to, or deletes an entry from, the Address Resolution Cache. **arp** with no parameters displays the contents of the Address Resolution Cache.

Syntax

arp add hostid ether | ieee hw_addr
arp drop hostid ether | ipx | ieee

Remarks

'arp add' includes a new entry in the Address Resolution Cache. Do not add entries with duplicate IP or hardware addresses as this will cause malfunctioning of the network.

arp drop deletes an entry from the Address Resolution Cache.

Options

hostid

The IP address of a remote host that is to be added to the ARP cache. This value may be a symbolic name from the HOSTS file or a decimal (dotted) address.

hw_addr

Used with add it denotes the hardware (node) address of the remote host whose IP address is given in *hostid*. This must be a six-digit hexadecimal address separated by colons for Ethernet.

Example Commands

```
arp add  unix_host  ether  00:00:65:0D:E6:04
arp add  127.0.1.3  ether  00:00:65:0D:E6:04
arp drop  192.6.1.12  ether
```

Example Output

```
ARP:   Received  1  BadType  0  BadAddress  0
       RequestsIn  1  Replies  0  RequestsOut  2
IP addr      Type            Time    Q Addr
192.6.1.4    10 MbEthernet   14      1 [unknown)
192.6.1.2    10 MbEthernet   891     00:00:d6:00:46:52
```

bootp

bootp changes the default retransmit time (5000 milliseconds) and timeout (30000 milliseconds) for **bootp** requests. A **bootp** request is made automatically as soon as the first **iface** statement is processed with a zero, or no, IP address. Consequently, the **bootp** specifications should occur before the **iface** statement. The **bootp** request is made after every retransmission time until timeout has elapsed. To cancel the **bootp** request sooner, press any key.

Syntax

bootp retransmission milliseconds
bootp timeout milliseconds

Remarks

bootp retransmission changes the default retransmit time value

bootp timeout changes the default timeout value.

Options

milliseconds

An integer value equal to the requested time in milliseconds.

Example Commands

```
bootp retransmission 5000
bootp timeout 30000
```

domain

If a host name is not a decimal (dotted) address and it is not found in the HOSTS file and at least one Domain Name Server has been defined, an attempt is made to obtain the address from the defined DNS server(s). The number of times any server is polled (retries), in addition to the time to wait for a response, can also be specified. A suffix may be specified and is attached to all names not containing any dots.

All of the following sub-commands can be issued without the optional parameters to obtain information on the current status.

Syntax

domain server [host_name]

domain retry [retry_count]

domain time [wait_time]

domain suffix [domain]

Remarks

domain server adds a DNS address or lists the current servers if host_name not specified.

domain retry specifies the retry count for polling each server. domain retry lists the retry count if retry_count not specified.

domain time specifies the time (milliseconds) to wait for a response before attempting retry. domain time lists the time (milliseconds) to wait if wait_time not specified.

domain suffix specifies the domain suffix to add to all simple names; names that contains no dots. domain suffix lists the domain suffix if domain is not specified.

Example Commands

```
domain retry 3
domain server 196.2.1.1
domain suffix myorg.co.za
domain time 2000
```

iface

iface is a synonym for the **interface command**.

interface

interface informs SOCKETS of the hardware or software communications interface(s) to be used at the network interface level. At least one network interface is required, and two or more are used in gateway (router) applications.

The class, or mode, of each interface defines the encapsulation used for packaging the data frame into the transport frame. Some types of interface support only one class.

When SOCKETS is defined with multiple interfaces, you first declare an IP address in the .CFG which is attached to the immediately following interface. The defined net mask is then used to add a route through the interface to the connected network. Using the same IP address would result in multiple routes to the same network. The default route is set on the first interface with an IP address with a zero net mask (for example, IP address 19.63.10.11/0).

Each interface statement uses the IP address from the last supplied IP address statement.

Syntax (general)

interface type name class other parameters

Syntax (specific)

interface pdr name dix mtu numbuf intvec [irq]

interface asy name [slip | cslip | ppp] mtu buflim ioaddr iovec speed [modemfile]

Options

type

type defines the type of hardware or software interface.

interface supports the following software interfaces.

Interface	Description
Asy	Standard PC asynchronous interface (RS232 port)
Pdr	packet driver interface

name

name defines the name by which the interface is known on the local host. *name* is a symbolic name known only to the local host on which it is used.

name may be arbitrarily assigned. Each interface command on the same host must have a unique name assigned. This name is used by commands such as route, trace, **par**, and so on.

class

class specifies how IP datagrams are to be encapsulated in the link level protocol of the interface. Some interfaces offer a choice between classes while others use a fixed class. The following classes are available and are listed with their associated types.

Type	**Class** (defined in the following list)
Pdr	dix, ieee, token, driver, slip
Asy	raw, slip, cslip, ppp

Class	Description
Dix	The DEC/Intel/Xerox Ethernet interface also known as Blue Book Ethernet or Ethernet II.
Token	IBM Token Ring. Source routing is supported for multiple rings.
Ieee	IEEE: 802.3 Ethernet with SNAP headers.
Driver	Use the default class for the packet driver.
Slip	Serial Link Internet Protocol (SLIP) for point-to-point asynchronous

Class	Description
	links. This mode is compatible with UNIX SLIP.
Cslip	Compressed Serial Link Internet Protocol (SLIP) for faster reaction over point-to-point synchronous links.
Ppp	Point-to-point protocol over asynchronous links.

mtu

mtu specifies the Maximum Transmission Unit size, in bytes. Datagrams larger than this limit are fragmented into smaller pieces at the IP layer. The maximum value of *mtu* for the various interfaces is:

Ethernet - 1500

For serial links a standard value for *mtu* is 576. (576 is the maximum according to specifications, but may be increased on reliable connections as long as both sides use the same value.)

numbuf

numbuf specifies how many incoming datagrams may be queued on the receive queue at one time. If this limit is exceeded, further received datagrams are discarded. This mechanism is used to prevent fast interfaces from filling up memory when data cannot be handled fast enough.

buflim

buflim specifies the maximum number of outgoing datagrams or packets to queue before starting to discard datagrams. This mechanism is used to prevent the memory from filling up when a serial link goes down.

bufsize

bufsize specifies the size of the ring buffer in bytes to be allocated to the receiver in raw mode

intvec

intvec specifies the software interrupt number (vector) in hexadecimal to use for resident packet drivers.

ioaddr

ioaddr is the I/O base address in hexadecimal of a serial port or the hardware controller and must correspond with the jumper or switch settings used during the setup of the controller board. The standard values for serial ports are:

COM1 03F8h

COM2 02F8h

COM3 03E8h

COM4 02E8h

iovec

iovec is the hardware interrupt vector used by the serial port or controller and must correspond with the jumper or switch settings used during setup of the controller. The standard values for serial ports are:

COM1 4

COM2 3

COM3 4

COM4 3

irq

irq is the hardware interrupt vector used by the network interface controller. This is only used for faster response in SOCKETS.

modemfile

A file containing the modem commands and scripts.

speed

speed specifies the transmission speed for serial interface devices (baud rate). Before using a serial connection you have to set flow control with the **par** command.

Examples

```
interface pdr if0 dix 1500 5 0x60
interface asy ser0 cslip 576 15 0x3f8 4 9600
interface asy p0 ppp 576 5 0x3f8 4 9600 pppmod.mod
```

IP

IP displays or sets the values of the options selected when defining the IP (internet protocol) host address of the next interface to be defined.

Syntax

IP address [hostid [/net_bits]]

IP status

IP ttl [number]

Remarks

IP address sets the IP host address of the next interface to be defined. A route is automatically added to each interface for the default or specified net mask for its address. To make an automatic route the default, specify the net bits as zero. When specified without the optional parameters, IP address displays the current value(s) of the local host IP address(es). To assign different IP addresses to different interfaces on the same host, an IP address statement must precede each interface definition. The last IP address given is used in case of missing IP address statements.

IP status displays Internet Protocol (IP) statistics, such as total packet counts and error counters of various types. It also displays statistics on the Internet Control Message Protocol (ICMP). This includes the number of ICMP messages of each type sent or received.

IP ttl sets the default time-to-live value which is placed in each outgoing IP datagram. The ttl value limits the number of gateway hops the datagram is allowed to take in order to kill datagrams that got stuck in loops.

Options

hostid

hostid specifies the IP host address to assign to the next interface to be defined. This may be a symbolic name from the HOSTS file, or a dotted decimal address.

/net_bits

A net mask can be specified for the host. In the **IP address** command an optional */net_bits* can be used to indicate the number of bits in the network ID. The net mask is used to determine whether an incoming datagram is a broadcast and also for sending UDP broadcasts.

Net masks are more easily represented in binary or hexadecimal format. For example, the IP address 128.1.1.5/24 corresponds to a net mask of 255.255.255.0 (FFFFFF00h), 25 bits to 255.255.255.128 (FFFFFF80h) and 26 bits to 255.255.255.192 (FFF FFC0h).

The default net mask used corresponds to the class of address used if not explicitly specified.

Net Bits	Net Mask	Class	IP address range
8	255.0.0.0	A	0.x.x.x to 127.x.x.x
16	255.255.0.0	B	128.x.x.x to 191.x.x.x
24	255.255.255.0	C and higher	192.x.x.x or higher

If you want to subdivide your network, you can divide it by two for every net bit added. The following table provides information on converting between net bits and net mask. The number of net bits to add when changing a 0 in the net mask to:

Net Bits	Net Mask	Net Bits	Net Mask
1	128	5	248
2	192	6	252
3	224	7	254
4	240	8	255

number

When *number* is omitted, **IP ttl** displays the current value of the time to live parameter.

par

par invokes a device-specific control routine. When executed without parameters, **par** displays defined interface names and device-specific flags. **par** operates differently for each interface type and even interface mode. In many cases it is used to query the status of an interface. The **ifstat** and **par** commands perform similar and, in some cases, exactly the same function.

Syntax

par ifname [arg1...argn]

Options

ifname

ifname defines the name used in the interface command for the device to be controlled.

arg1...argn

These parameters depend on the type of interface in use.

Example

To display current serial link settings and restart the statistics on it, use:

```
par sl0 clear
```

par, Alternative Routing Control Sub-commands

The Alternative Routing Control Sub-commands set up and check the SOCKETS alternative route mechanism. More than one route can be specified to a target host or network. The first route that has an associated interface in the up state is used.

An interface is in the up state when it is defined by the interface command. It enters the query state when it does not receive valid input within a specified up-time period after data expecting a response is

sent.. At this stage three (catering for links with a high data loss) ICMP echo requests (ping) are sent to a query IP address. It enters the down state by a SOCKETS command or when it does not receive valid input within the specified up-time period after entering the query state. If an up time has never been specified or a value of 0 is specified, the interface will stay in the up state whether valid input is received or not.

An interface enters the up state by a SOCKETS command or when valid input is received on that interface while it is in the down or query states. An ICMP echo request is sent on an interface in the down state every downtime period. If a downtime has never been specified or a value of 0 is specified, the ICMP echo request is not sent. Up time and downtime is specified in seconds.

Syntax

par ifname [uptime | downtime] time

par ifname query hostname

Options

Ifname

Ifname is the interface name of an asy interface.

Uptime

Uptime is the time to allow for no response on a defined connection.

Downtime

Downtime is the period of time to retry a defined connection.

Example Alternative Routing

Two X.25 interfaces are used to get to the target network 192.6.1.0. The first interface, named if0 should preferably be used, but if it stops receiving for a period of 20 seconds, it should try to ping 192.6.1.2 and if no response is received within another 20 seconds, if1 should take over, but if0 should be tried every five seconds. Interface if1 should disconnect after 80 seconds of no traffic.

The SOCKET.CFG file should contain the following:

```
interface x25 if0 ...  ...  ...
par if0 uptime 20
par if0 downtime 5
par if0 query 192.6.1.2
interface x25 if1 ...  ...  ...
par if1 uptime 80
par if1 downtime 5
par if1 query 192.6.1.2
route add 192.6.1.0 if0
route add 192.6.1.0 if1
```

In the case of both if0 and if1 failing, both will retry every five seconds until one comes up. The return paths should also be maintained in a similar way with SOCKETS or by using RIP.

par, COM Port Speed and Flow Control Sub-command

This par sub-command allows the baud rate, flow control and data parameters to be set and is only to be used on asy type interfaces.

Syntax

par ifname speed outflow inflow bits

Options

ifname

ifname is the interface name of an asy interface.

speed

speed sets the baud rate for a serial link. The standard speeds are: 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400 and 57600.

outflow

outflow sets the output flow control for a serial link. When not supplied it defaults to none.

inflow

inflow sets the input flow control for a serial link. When not supplied it defaults to none. The following list shows *outflow* and *inflow* selections.

none	no flow control
Xon	Xmit on/Xmit off (Ctrl-Q, Ctrl-S)
DCD	Data Carrier Detect modem signal
CTS	Clear To Send modem signal
DSR	Data Set Ready modem signal
DTR	Data Terminal Ready modem signal
RTS	Request To Send modem signal
ixxx	inverse of modem signal xxx

Invalid selections are ignored or replaced by more logical selections. The usage of Xon/Xoff is not recommended for most applications. The ixxx support is for non-standard equipment that uses a reversed signal on the required pin.

bits

bits sets the number of data bits per character, number of stop bits, and parity for a serial link. It is a three-character string consisting of dsp from the following table. When not supplied, *bits* defaults to value of 81n.

Values for d	5, 6, 7, or 8 data bits
Values for s	1 or 2 stop bits
Values for p	o = odd parity e = even parity n = no parity m = mark s = space

Example

```
par  asy1  57600 cts rts 81n
par  slp0  4800 xon xon 72e
```

par, RIP Advertising Sub-command for Interfaces

When the RIP advertise command has been used, this par sub-command makes allowance to disable and re-enable RIP advertising on a specific interface.

Syntax

par ifname [ripadv | noripadv]

Options

Ifname

ifname is the interface name of an asy interface.

Ripadv

Ripadv indicates to use route advertising on the defined interface.

Noripadv

Noripadv indicates to not use route advertising on the defined interface.

Examples

```
par if0 noripadv
par if1 ripadv
```

printer

The **printer** redirector operates by intercepting Interrupt 17 and passing printer output to a local or remote print server. A print session is started when output is first sent to a specific port, and is stopped after a user-specified period of no output. When a printer port is redirected, the IP address and TCP port number of the destination print server, in addition to the timeout period, is specified. A printer port can both be used as a print server port and as a redirected port.

Syntax

printer printer_port timeout IP_address [TCP_port]

Options

printer_port

Printer port 0, 1, 2 or 3 to redirect. Port 0 corresponds with PRN, 1 with LPT2 and so on. Serial printers can be defined with the SRPRINT.EXE TSR that forms part of the SOCKETS package.

timeout

Timeout period in seconds for closing the TCP connection.

IP_address

IP address of print server host (could be the local host).

TCP_port

TCP port of print server (Default is 10).

Comments

To ensure proper queuing of remote and local print sessions to a local printer, the printer port must be redirected to the local IP address and TCP port of the print server using that port.

Examples

The IP address of print_host in the following example is defined in the HOSTS file.

```
printer 0 15 print_host
```

The following example uses printer port 1 (LPT2) for both local and remote printing. (If the printer command is placed before start prntserv, SOCKETS will not work correctly.)

```
ip address this_host
 .
 .
start prntserv 1010 1
printer 1 15 this_host 1010
```

RIP

The Routing Information Protocol (RIP) allows a SOCKETS gateway to advertise routes and allows SOCKETS to recognize advertised routes from SOCKETS and other gateways. These two facilities can be individually selected.

Syntax

RIP advertise [time [1|2] [self]]

RIP use [time]

Options

time

time is the number of elapsed seconds before the advertisement is repeated when used in the advertise sub-command.

time is the period during which routes are added or amended as a result of RIP, and are valid for *time* seconds. Such added or amended routes are dropped if another RIP advertisement is not received within *time*.

self

self advertises SOCKETS connection to the network.

RIP Advertise Sub-command

RIP advertise causes an immediate advertisement of all relevant routes and repeats it every time seconds. The default value for time is 30 seconds. RIP version 2 advertisements are sent by default. To send only version 1 advertisements, add a 1 on the command line.

When RIP advertise is selected, all interfaces advertise all routes except those routes making use of that specific interface (split horizon) and routes marked Private. (To prevent certain interfaces from using RIP, see the parameter command.) A route which is dropped as a result of a RIP update or which becomes unavailable as a result of its associated interface going into the down state, is immediately advertised as being infinite (metric = 16) and is not advertised until it becomes available again. The advertisement is sent as a sub-net broadcast using the net mask and IP address of the interface.

Example

```
RIP advertise 30 self
```

RIP use, Update RIP Sub-command

RIP use causes a RIP request for routes and a continuous update of routes according to RIP advertisements. Routes added or amended as a result of RIP are valid for time seconds and are dropped if another RIP advertisement is not received within that time. The default value of time is 240 seconds.

When RIP use is selected, routes are updated according to received RIP advertisements. Routes added or amended as a result of RIP, have a timeout associated with them. If another RIP advertisement is not received during that time, the route is dropped. A route is also dropped if an advertisement of infinity (metric = 16) is received. To prevent dropping a route, it must be marked as Static.

On RIP and static routes: — The metric of a route marked static is never updated by a RIP advertisement. Instead a duplicate route is added before the static route. If the duplicate route is dropped as a result of a timeout or RIP, the static route is used again.

On RIP and mdd or x25 interfaces: — On mdd or x25 interfaces, a RIP route is only sent when the last datagram was not a RIP route. This is done to allow an interface to time out and be disconnected when not used.

Also, the route on the receiving end still times-out and becomes unavailable. This means that inactivity timers should be shorter than route timers; as specified in RIP use timer.

When an aslink to mdd association is broken, all non-static routes using the specific mdd are marked as having a hop-count of infinity; the metrics are set to 16.

route

route creates an entry in the IP routing table for SOCKETS to determine where to send data. The Alternative Routing mechanism allows more than one route to be specified to a particular host or network. Failure of one route causes an automatic switch to the next route.

Refer also to the **ip** address command for specifying the net mask, because a route is automatically added to each interface for the default or specified net mask for that address. When multiple routes are defined to the same address, SOCKETS uses the route with the network size (largest number of bits in the net mask).

Syntax (general)

route [add | drop destination ifname [gateid |none [metric [proxy] [private] [static]]]]

Syntax (specific)

route

route add [hostid | netid] ifname [gateid]

route add [hostid | netid[/mask]] ifname [gateid]

route add default ifname

route drop [hostid | netid]

route drop [hostid | netid[/mask]]

route drop default

Options

add or drop

Sub-command to add or drop (remove) a route from the routing table.

default

All transmissions to IP addresses not otherwise defined in routing commands are sent via the network interface specified by *ifname*.

hostid

hostid is the IP address of a destination remote host to which data must be sent, or a remote host that must be removed from the routing table (dropped).

netid

netid is the IP address of a destination network to which data must be sent. Any host with this IP network address is able to receive the data. Whether a particular host will use the data depends on the host portion of the specific IP address in the IP header.

mask

mask specifies the number of bits in the network portion of the address if sub-netting is used. If not used, the network portion of the address is determined according to the class (A, B or C) of the address.

ifname

ifname defines the name used in the interface command for the immediate network on which the data for the designated host must be sent. This is the network level interface to be used by the local host to reach the remote host.

gateid

gateid parameter specifies the IP address of a host, on the same physical network as the local host, which is used as a gateway or router to a different network. The gateway or router host specified in *gateid* must be directly reachable on the same physical network as the local host defining this gateway. In other words, this must be the nearest gateway to this local host.

metric

When using RIP or Proxy ARP a value from 0 to 16 for *metric* must be specified indicating the distance or cost of that route. A *metric* of 16 indicates that the route is down.

proxy, private and static

To support the Routing Information Protocol (RIP) the route command utilizes the proxy, private and static key words. These words can be used in any order following *metric*.

Proxy ARP should be used with care and not in conjunction with RIP. When more than one host responds to an ARP request, it can cause confusion and even lead to system crashes. This is possible in situations where more than one gateway implements Proxy ARP to a common destination. See also “Address Resolution Protocol (ARP)” on page 6.

When “RIP advertising” is selected, all interfaces advertise all routes except those routes making use of that specific interface (split horizon) and routes marked private. A route which is dropped as a result of a RIP update or which becomes unavailable as a result if its associated interface going into the down state, is immediately advertised as being infinite (metric = 16) and is not advertised until it becomes available again. In order for an interface to be used for advertising, a route without a gateway using that interface must be available. The advertisement is sent as a sub-net broadcast using the net mask of the host and the IP address of the interface.

When “RIP using” is selected, routes are updated according to received RIP advertisements. Routes added or amended as a result of RIP, have a timeout associated with them. If another RIP advertisement is not received during that time, the route is dropped. A route is also dropped if an advertisement of infinity (metric = 16) is received. To prevent dropping a route, it must be marked

as static. The metric of a route marked static is never updated by a RIP advertisement. Instead a duplicate route is added before the static route. If the duplicate route is dropped as a result of a timeout or RIP, the static route is used again.

Examples

```
route add default ipx0
route add unix_net eth0
route add unix_host ipx1 unx_gate
route add unix_net2 eth0 /eth 1
route add unix_ net ipx0 unx_ gate
route add subnet/26 eth0 sub_gw
route drop unix_net
```

route can specify a Proxy ARP on a route, as follows:

route add net interface gateway metric [proxy]

When using Proxy ARP, gateway and metric must be specified. If no gateway is used, none can be specified. For example:

```
route add 192.6.1.0 ifx25 none 5 proxy
```

start

start starts a SOCKETS server (prntserv)and makes it available for access by remote hosts.

Syntax

start prntserv [port [printer_number]] [/m=mask] [/s=status] [/r=report]

Options

port

port is the TCP port to use (the default is 10)

printer_number

printer_number is a value of 0, 1, or 2 to correspond with LPT1(PRN), LPT2 and LPT3. Any number up to 3 may be defined by the serial printer TSR (SRPRINT) as a printer on a COM port. The default printer number is 0.

mask

Used to activate status reporting and cope with non-standard printer status reporting.

status

Used to activate status reporting and cope with non-standard printer status reporting.

report

Used to activate status reporting and cope with non-standard printer status reporting.

Printer Status Reporting

Optional status reporting has been implemented to give print client implementations more control over print jobs. The following print server enhancements have been enabled:

1. Optional status reporting to Print Client. The status reporting includes such messages as:

   Timeout
   Last character has been sent to printer
   I/O Error

Printer selected
Out of paper
Acknowledged
Printer Ready

2. Customization of printer status checking.

3. Print queue flushing.

Raw printer status bytes as defined for BIOS INT 17 services can be sent from the server to the client on the established connection when:

1. The first data is being printed.

2. The status changes while attempting to print.

3. The last character in the stream (the character before the FIN) has been sent to the printer. The unused bit Bit1 in the status is used to indicate this event.

The status byte is defined as follows:

Bit7	80h	Printer ready
Bit6	40h	Acknowledged
Bit5	20h	Out of paper
Bit4	10h	Printer selected
Bit3	08h	I/O Error
Bit2	04h	Unused
Bit1	02h	Last character has been sent to printer
Bit0	01h	Timeout

The status bits are returned by the Int 17 BIOS call (with the exception of Bit1) and are not always consistent, but depend on the BIOS of the particular server.

The utility PRNTEST.EXE in the TEST subdirectory checks the response from the selected parallel port where LPT1 = 0; LPT2 = 1 and LPT3 = 2. Start PRNTEST.EXE 0 to test LPT1 for example. PRNTEST shows the returned BIOS code from the printer port.

To print successfully, three conditions must be met: 1) the printer must be selected, 2) be ready, and 3) contain paper.

Sum the hexadecimal values for these conditions and use that as the /s and /m parameters for the start prntserv commands. The default values used are /mA8 for the mask and /s80 for the status which implies that data is sent to the printer when it is not busy and out-of-paper, or I/O Error status is not set. Printer selected is ignored as this status bit is often not reported correctly. To include the printer selected bit, specify /mB8 and /s90.

start prntserv, Socket Print Server

The socket print server accepts TCP connections and routes output to one of four system printers on a parallel port or serial port. Up to four print servers can be started on different TCP ports, connecting to different printers. The server accepts multiple calls, but prints strictly in order of received calls. This means that a connection must be closed to allow another client to print, thereby providing print queuing. There is no indication of queuing status, so a non-ready printer will not hang SOCKETS. Other operations are not affected.

The first entry on the print queue can be flushed to allow printing to continue.

Because prntserv is not a telnet server, any telnet commands are passed as data.

Syntax

start prntserv [port [printer_number]] [/r=report] [/m=mask] [/s=status]

Example

```
start prntserv 10 0
```

Activate Printer Status Reporting

The print server is instructed to send status bytes in the start prntserv command by specifying an optional argument /r=report where report is a hexadecimal value specifying the status bits which are checked for changes to initiate a report. (A report is always sent at the beginning of a job.) The default value is 0 that means that the print server will not report any status changes. A logical value to use is /r=3F which causes any changes, except Printer ready and Acknowledged, to be reported. (The Acknowledged bit is not useful in this application because it toggles at each character; too fast for most networks to carry all the generated traffic.) Note that the full printer status byte is transmitted without filtering any bits.

The start prntserv command can also cater for non-standard status responses from printers. Two additional arguments may be specified directly following prntserv and before the optional port. They are /m=mask where mask is the hexadecimal value of the mask to be applied to the status indication from the printer and /s=status where status is the hexadecimal value of the resultant status which causes the printer to print. These options are only used locally at the server and the result is not passed on to the client. Change these options only when error conditions (for example, printer switched off or printer off-line) causes output to be lost. The default values used are 0xA8 for the mask and 0x80 for the ready status which implies that data is sent to the printer when it is not busy and Out of paper or I/O Error status is not set. Note that Printer selected is ignored as this status bit is often not reported correctly. To include the Printer selected bit, specify /m=B8 and /s=90.

The SOCKETS utility PRNTEST.EXE can be used to test your parallel port.

stop

stop stops the specified server, rejecting any further remote connect requests. Existing connections are allowed to complete normally. A special case is the flushing of the socket printer, where the server continues with the next queued job after flushing the first entry. To determine which servers are currently running, execute the TCP Status command and look at the local SOCKETS.

Syntax

stop server

stop prntserv PCprinter reset

Options

server

This parameter specifies the name of the server to be stopped. The choices are specified under the parameter heading in the description of the start command.

PCprinter reset

The printer port number (0 to 2 for LPT1 to LPT3, or 3 for the fourth printer) to flush.

Examples

```
stop discards
stop echoserv
```

```
stop FTPserv
stop prntsrv Pcprinter reset 0
```

tcp

tcp commands display or set various TCP operating parameters. The TCP configuration commands are put into SOCKETS.CFG.

Syntax

tcp irtt [time]

tcp lport [port_number]

tcp mss [size]

tcp retry [number]

tcp rtt [time]

tcp smss [size]

tcp timemax [time]

tcp window [size]

Options

time

time is the new time value in seconds, or milliseconds if "ms" is appended to the number, as in 2000ms.

port_number

port_number is the local port starting number.

size

For **tcp mss**, *size* is the maximum segment size in bytes sent on all outgoing TCP connect requests (SYN segments). *size* tells the remote host the size of the largest segment that may be received by this host. When changing the MSS value, any existing connections remain unchanged.

For **tcp smss**, *size* is the send maximum segment size in bytes sent on all outgoing TCP connect requests. This limits the size of the largest segment that may be sent by this host. When changing the SMSS value, any existing connections remain unchanged.

For **tcp window**, *size* is the size of the receive window in bytes for any new TCP connections. Existing connections are unaffected.

number

number is the number of retries attempted without receiving an acknowledgement from the remote host before the connection is broken. If the value exceeds 255, it implies an infinite number of retries; such a connection does not time-out. The default value for *number* is 6.

tcp irtt Sub-command

tcp irtt displays or sets the initial round-trip-time estimate. When specified without an argument, the command displays the current values of TCP parameters including the initial round-trip-time in milliseconds.

time is the initial round-trip-time (IRTT) estimate and is used for new TCP connections until the actual value can be measured and adapted to. By increasing this value when operating over slow communication links, unnecessary retransmissions that otherwise occur before the smoothed estimate value approaches the correct value are minimized. The system default is 5000 milliseconds.

To affect incoming connections, tcp irtt should be executed before the servers are started.

Example

```
tcp irtt 120
```

Sample Output

```
TCP: IRTT 5 ms Retry 6 MSS 1460 SMSS 1460 Window 2920
```

tcp lport Sub-command

tcp lport specifies the local port starting number. When specified without a number the current value of the next free local port number is displayed.

Example

```
tcp  lport  2004
```

Sample output

```
lport = 2004
```

tcp mss Sub-command

tcp mss displays or sets the TCP maximum segment size in bytes. When size is not specified, the current values of the TCP parameters, including the maximum segment size, are displayed. It is recommended to reduce the MSS and SMSS on bad network connections. The SOCKETS queuing overhead is 12 bytes per WriteSocket request i.e. you will need 52 bytes for each request. It will queue 2*MSS bytes for you if it can't send right away because of window constraints by the peer or the Nagle heuristic. That means that if your MSS is set to the default of 1460, 2920 bytes will be queued, consuming 3796 bytes in your case. If you have 5 connections, you need 18980 bytes just to buffer your outgoing data. Use IPSTAT to determine how much memory you have available for everything. (You need memory for each connection, incoming data and all other TCP functions).

Example

```
tcp mss 1460
```

tcp retry Sub-command

tcp retry displays or sets the retry count before a connection is broken. When specified without the number parameter, tcp retry displays the current values of TCP parameters, including the retry count. Refer also to “TCP Retry Strategy” on page 179.

tcp rtt Sub-command

tcp rtt replaces the automatically computed round-trip time (RTT) for the specified connection with the time in milliseconds. SOCKETS calculates the RTT as a smooth average of past measured RTTs, starting with the IRTT on a new connection. To get the current RTT in use for a connection n, use the tcp status n command that gives the smoothed average RTT indicated by SRTT. Because tcp rtt

provides a manual override of the normal back-off retransmission timing mechanisms, it may be used to speed up recovery from a series of lost packets.

Example

```
tcp rtt 4 100
```

tcp smss Sub-command

tcp mss displays or sets the TCP send maximum segment size in bytes. When size is not specified, the current values of the TCP parameters, including the SMSS, are displayed. A small SMSS causes the remote to reduce its segment size. tcp mss can reduce the MSS and SMSS on bad network connections with high loss rates or where large packets get lost.

Example

```
tcp smss 512
```

tcp window sub-command

tcp window displays or sets the default and maximum receive window size. When specified without the size parameter the current TCP parameters, including the current window size, are displayed.

Example

```
tcp window 2920
```

tcp timemax Sub-command

tcp timemax sets the maximum duration of a tcp retry. If a value greater than 255 seconds is specified, connections never timeout. This is very useful in wireless applications where nodes roam in and out of service.

Example

```
tcp timemax 2000ms
```

Chapter 7, SOCKETS Applications (Application Layer)

General Application Notes

SOCKETS applications make use of command-line parameters and/or configuration files. Please be careful to note the name and location of the configuration file used by the application you are working with. All SOCKETS applications require that the kernel be loaded before the application is run in order to function properly.

E-mail Applications

These applications allow a DOS platform to send and receive Internet e-mail from a SOCKETS host.

MAKEMAIL

MAKEMAIL packages the body text and any attachments for delivery using the **SENDMAIL** application.

Syntax

MAKEMAIL –tToAddress –fFromAddress –sSubject –bBodyTextFile -oOutputFile-aAttachment

Options

ToAddress

The e-mail address of the recipient(s) of this mail. Additional recipients are specified by repeated use of the **–t** parameter. If the *ToAddress* is a name that can be resolved by either the DNS server or host file then the *@servername* is not necessary.

FromAddress

Used to identify the sender of the message.

Subject

The subject line of the e-mail message.

BodyTextFile

The local file containing the body text of the e-mail message to deliver.

OutputFile

The local file name in which to store the prepared file for delivery by **SENDMAIL**. This file is overwritten if it already exists!

Attachment

The name of a local file to be binary attached to this e-mail message. Multiple attachments are created by repeated use of the **–a** parameter. Files are attached as MIME parts, encoded with the application/x-uuencode content type.

Example

```
MAKEMAIL -tfred@yahoo.com -fmary@yahoo.com -sStatus -bmessage.txt -omail.dat
MAKEMAIL -tfred -tbarney -fwilma -sDinner -bmenu.txt -omail.dat
MAKEMAIL -tfred -fwilma -sBowling -bbody.txt -aStone.jpg -aRock.jpg -omail.dat
```

Example Output

MAKEMAIL does not output status information.

SENDMAIL

SENDMAIL delivers e-mail messages packaged by the **MAKEMAIL** application to an Internet mail server. **SENDMAIL** also creates a local log file to indicate successful send or failures.

Syntax

SENDMAIL server file

Options

Server

The IP address or DNS name of the Internet mail server to receive the message.

File

The file, created by the **MAKEMAIL** utility, to deliver.

Logging Format

Timestamp, Code String

Timestamp

Weekday Month Day Time Year

Code

Three digit integer. 000 means perfect success, 100-199 mean usage error and 200-299 means TCP/IP error from server.

String

Human–readable explanation of the error code.

Example

```
SENDMAIL mail.datalight.com mail.dat
```

Example Output

```
Mon Jan 03 14:39:21 2000, 100 SENDMAIL <server> <file>
Mon Jan 03 14:39:21 2000, 000 Sent SMTP mail successfully
```

GETMAIL

GETMAIL retrieves all of the messages from a POP3 (Post Office Protocol version 3) Internet mail server. Each message is stored as an individual file on the local machine. *GETMAIL* also creates a log file to indicate successful downloads or errors.

Syntax

GETMAIL server user password

Options

Server

The IP address or DNS name of the Internet mail server from which to download messages. The messages that are downloaded are named by sequential file number and are placed in the current working directory.

User

The username for server identification purposes.

Password

The secret for account authentication on the server.

Logging Format

Timestamp, Code String

Timestamp

Weekday Month Day Time Year

Code

Three digit integer. 000 means perfect success, 100-199 mean usage error and 200-299 means TCP/IP error from server.

String

Human–readable explanation of the error code.

Example

```
GETMAIL 10.0.0.1
GETMAIL 10.0.0.1 guest secret
```

Example Output

```
Mon Jan 03 14:39:21 2000, 100 GETMAIL <server> <user> <password>
Mon Jan 03 14:39:21 2000, 000 Retrieved POP3 mail successfully
```

HTTP Applications

These applications allow serving and retrieving files via the standard HTTP protocol, and also allow remote connection to the host's console.

HTTPD

HTTPD is a web server that can run either as an application or as a TSR. In addition to processing normal HTTP requests on default port 80, it serves a proprietary session displaying the contents of text-mode display memory to the **RC.JAR** and **RCCLI** client applications on default port 81. This feature is commonly called the "remote console."

If the **HTTPD** web server is loaded as a DOS TSR program, set the environment variable, HTTP_DIR, to the location of the INDEX.HTML file; for example, SET HTTP_DIR=C:\DL\SOCKETS\SERVER

Syntax

HTTPD [options] [<http_port>] [<rc_port>]

Options

/a=ScreenX,ScreenY

/v=ScreenBufferSegment[:ScreenBufferOffset]

/m=MemorySize

/n=MaximumConnections

/i=InterruptNumber

/d

/c

/t

/r

/s

/u

Remarks

ScreenX, ScreenY

The width and height of the screen area to serve for the remote console session. These values default to 80 and 25, respectively.

ScreenBufferSegment, ScreenBufferOffset

Together, a pointer to the top-left corner of the display memory to serve for the remote console session. These values default to B000 and 0000 respectively, for monochrome display adapters and to B800 and 0000 respectively, for color display adapters.

MemorySize

The maximum amount of memory available to the server. The default value is 32K. The value of m can range from 8192 to 63472.

MaximumConnections

The maximum number of simultaneous connections allowed by the server.

InterruptNumber

The interrupt number to access the CGI API.

/d

When loading, don't start remote console.

/c

When in TSR mode, running **HTTPD** again with this option disables the TSR's listen socket.

/t

Enables htaccess that contains authentication overrides to enable partial anonymous access or additional password security to subdirectories. Please refer to Chapter 10 for further details and examples.

/r

Load as a TSR.

/s
When in TSR mode, running **HTTPD** again with this option displays the TSR's status.

/u
When in TSR mode, running **HTTPD** again with this option unloads the TSR.

http_port
HTTP port to listen on. This parameter defaults to the standard HTTP port number of 80.

rc_port
Remote Console port to listen on. This parameter defaults to 81.

Configuration File

HTTPD uses the standard SOCKET.UPW file for validating logins. The file is composed of text lines, each representing a login name, password, and the configuration to use for a session opened with those credentials. Space characters separate the parameters in the file, which are in the following format:

name password directory rights

The location of the username/password file to be used by the server is specified by the environment variable SOCKETS as follows:

%SOCKETS%\SOCKET.UPW

If the variable SOCKETS is not specified, the following file is used:

\DL\SOCKETS\SOCKET.UPW

Configuration File Parameters

name
The login name of this record.

password
The password to authenticate a user trying to login as this name.

directory
The starting directory for this user.

rights
Up to four characters specifying which permissions this user is granted:

e means that this user may 'g**e**t' files

p means that this user may '**p**ost' files

g means that this user may use c**g**i

m means that this user may use Re**m**ote Console

Example

```
HTTPD /m=40000 /r
HTTPD /a=80,25 /v=a000:0000 /r
```

Example Output

```
Sockets HTTP/RC Server V7.10 (Revsion 4.10.1154)
Copyright © 1989-2001 Datalight, Inc.
Portions copyright © GPvNO 2001
(compiled: 06/27/2001)

Going resident
```

HTTPFTPD

HTTPFTPD is a combined HTTP and FTP server that can run either as an application or as a TSR. By default, it processes normal HTTP requests on port 80 and normal FTP requests on port 21. It also serves a proprietary session displaying the contents of text-mode display memory to the **RC.JAR** and **RCCLI** client applications. This feature is commonly called the "remote console."

If the **HTTPFTPD** server is loaded as a DOS TSR program, set the environment variable, HTTP_DIR, to the location of the INDEX.HTML file; for example, SET HTTP_DIR=C:\DL\SOCKETS\SERVER

Syntax

HTTPFTPD [options] [<http_port> [<ftp_port> [<rc_port>]]]

Options

/a=ScreenX,ScreenY

/v=ScreenBufferSegment[:ScreenBufferOffset]

/m=MemorySize

/n=MaximumConnections

/i=InterruptNumber

/d

/p

/c

/t

/r

/s

/u

Remarks

ScreenX, ScreenY

The width and height of the screen area to serve for the remote console session. These values default to 80 and 25, respectively.

ScreenBufferSegment, ScreenBufferOffset

Together, a pointer to the top-left corner of the display memory to serve for the remote console session. These values default to B000 and 0000 respectively, for monochrome display adapters and to B800 and 0000 respectively, for color display adapters.

MemorySize

The maximum amount of memory available to the server. The default value is 32K. The value of m can range from 8192 to 63472.

MaximumConnections

The maximum number of simultaneous connections allowed by the server.

InterruptNumber

The interrupt number to access the CGI API.

/c
When in TSR mode, running **HTTPFTPD** again with this option disables the TSR's listen socket.

/t
Enables htaccess that contains authentication overrides to enable partial anonymous access or additional password security to subdirectories. Please refer to Chapter 10 for further details and examples.

/r
Load as a TSR.

/s
When in TSR mode, running **HTTPFTPD** again with this option displays the TSR's status.

/u
When in TSR mode, running **HTTPFTPD** again with this option unloads the TSR.

/p
Start **HTTPFTPD** in passive mode.

/d
When loading, don't start remote console.

http_port
HTTP port to listen on. This parameter defaults to the standard HTTP port number of 80.

ftp_port
FTP port to listen on. This parameter defaults to the standard FTP port number of 21

rc_port
Remote Console port to listen on. This parameter defaults to 81.

Configuration File

HTTPFTPD uses the standard SOCKET.UPW file for validating logins. The file is composed of text lines, each representing a login name, password, and the configuration to use for a session opened with those credentials. Space characters separate the parameters in the file, which are in the following format:

name password directory rights

The location of the username/password file to be used by the server is specified by the environment variable SOCKETS as follows:

%SOCKETS%\SOCKET.UPW

If the variable SOCKETS is not specified, the following file is used:

\DL\SOCKETS\SOCKET.UPW

Configuration File Parameters

name
The login name of this record.

password
The password to authenticate a user trying to login as this name.

directory
The starting directory for this user.

rights

Up to four characters specifying which permissions this user is granted:

e means that this user may 'g**e**t' files

p means that this user may '**p**ost' files

g means that this user may use c**g**i

m means that this user may use Re**m**ote Console

r means that this user has read access.

w means that this user has write access.

c means that this user has permission to make new directories.

d means that this user has permission to change to a directory other than his starting location and subdirectories from the starting location.

Example

```
HTTPFTPD /m=40000 /r
HTTPFTPD /a=80,25 /v=a000:0000 /r
```

Example Output

```
Sockets FTP/HTTP/RC Server V7.10 (Revsion 4.10.1154)
Copyright © 1989-2001 Datalight, Inc.
Portions copyright © GPvNO 2001
(compiled: 06/27/2001)

Going resident
```

HTTPGET

HTTPGET is a simple web client that can retrieve the contents of a URL to a local file.

Syntax

HTTPGET [-p] [–s] [–v]URL

Options

-p=Port

-s=Server

-v

localfile

Remarks

Port

Use to specify a remote port other than 80 to connect to.

Server

Use to specify a server name if the URL doesn't contain one.

-v

Display extra output for troubleshooting.

localfile
Rather than keeping the filename from the URL, the contents may be saved to a named file.

Example
```
HTTPGET http://www.datalight.com/images/logohead.gif
HTTPGET -v http://www.datalight.com/images/logohead.gif logo.gif
```

Example Output
```
HTTPGET version 1.0
```

RC.JAR

RC.JAR is a Java remote console applet for connecting to hosts running **HTTPD**. It shows the text-mode contents of the **HTTPD** machine's display memory, and allows fully interactive keyboard input. This is an alternative to the **RCCLI** applet.

Syntax
The link to RC.JAR should be embedded in an HTML page served by the HTTPD server. An example is available in the CGI/ subdirectory of the HTTP applications.

Remarks
RC.JAR has been compiled for use with Microsoft's Internet Explorer. In order to use with a Netscape Browser a security certificate must be compiled into RC.JAR.

RCCLI

RCCLI is a DOS remote console client for connecting to hosts running **HTTPD**, and is an alternative to the **RC.JAR** applet.. It shows the text-mode contents of the **HTTPD** machine's display memory, and allows fully interactive keyboard input. To exit the remote console client, press Ctrl-Alt-X.

Syntax
RCCLI server_address <tcp_port>

Options
server_address
Specify the IP address or DNS name of the machine running **remcon**.

tcp_port
tcp_port defaults to 81, but must be changed if a nonstandard remote console port is chosen for the remote console session during server configuration.

Example
```
RCCLI 10.0.0.1
```

Example Output
```
An interactive image of the remote DOS session.
```

FTP Applications

These applications allow serving and retrieving files via the standard FTP protocol, either from the command line or from within an application.

FTPD

FTPD is a file server that can run either as an application or as a TSR. The name of the server as displayed in the banner is determined by the HOSTNAME environment variable. If the environment variable is not set, the name "Socket" is used. The user password file, SOCKET.UPW, in the SOCKETS directory (indicated by the SOCKETS environment variable) controls access.

A temporary file is created when a directory listing is requested. This file is created in the current directory, but can be created in any directory as specified in the FTPDIR environment variable.

Syntax

FTPD [options] [<ftp_port>]

Options

/m=MemorySize

/n=MaximumConnections

/c

/r

/s

/u

Remarks

MemorySize

The number of bytes of memory available to the server. This value defaults to 32768.

MaximumConnections

The maximum number of simultaneous connections allowed by the server.

/c

When in TSR mode, running **FTPD** again with this option disables the TSR's listen socket.

/r

Load as a TSR.

/s

When in TSR mode, running **FTPD** again with this option displays the TSR's status.

/u

When in TSR mode, running **FTPD** again with this option unloads the TSR.

ftp_port

FTPD will listen on the listed port. This parameter defaults to the standard FTP port number of 21.

Configuration File

FTPD uses the standard SOCKET.UPW file for validating logins. The file is composed of text lines, each representing a login name, password, and the configuration to use for a session opened with those credentials. Space characters separate the parameters in the file, which are in the following format:

name password directory rights

The location of the username/password file to be used by the server is specified by the environment variable SOCKETS as follows:

%SOCKETS%\SOCKET.UPW

If the variable SOCKETS is not specified, the following file is used:

\DL\SOCKETS\SOCKET.UPW

Configuration File Parameters

name

The login name of this record.

password

The password to authenticate a user trying to login as this name.

directory

The starting directory for this user.

rights

Up to four characters specifying which permissions this user is granted:

r means that this user has read access.

w means that this user has write access.

c means that this user has permission to make new directories.

d means that this user has permission to change to a directory other than his starting location and subdirectories from the starting location.

Example Socket.upw

```
Admin admin drwc
Guest * c:\guest dr
```

Example

```
FTPD /m=40000 /r
```

Example Output

```
Sockets FTP Server V7.10 (Revsion 4.10.1154)
Copyright © 1989-2001 Datalight, Inc.
Portions copyright © GPvNO 2001
(compiled: 06/27/2001)

Going resident
```

FTP Server Commands

The following commands are recognised by the SOCKETS FTP server:

abort cancel an incomplete transfer
append "put" a file at the server but append it if the file exists
cwd *directory* change server directory
dele *file* delete a server file
list [*file* l *directory*] give a long directory listing
mkd *remote_directory* create a server directory
nlst [*file* l *directory*] gives a short names-only directory listing
pass [*password*] password for username
pasv [on | off] report or change the status of the passive transfer mode to enable firewall friendly file transfers. (The SOCKETS FTP client always tries to switch passive mode on at the start of a session.)
retr *remote_file* transfer a file from the server in the current mode
stor *local_file* transfer a file to the server in the current mode
pwd print working directory
quit terminate FTP session
rmd *remote_directory* remove (delete) directory
rnfr *existing_filename* rename a file, command 1 of 2
rnto *new_filename* rename a file, command 2 of 2
site [path I nopath] use full path description (see
site raw [*interface*] open a session to a raw host using one of the raw lines (interfaces) specified
site *sub-command* command to be passed on to raw host
size *file* report the file size in bytes as a message prefixed with 213
stat report the status of a transfer or active connections
system return operating system information from the server
type [i I a] report or select the file transfer mode: image (binary) or ASCII
user [*username*] username to logon

FTP

FTP is a file transmitting and retrieving client that runs in interactive or batch mode.

Syntax

FTP server [options]

Options

/n

/v

/p=Port

/f=ScriptFile [ScriptParameters]

Remarks

Server
The name or ip address of a server to connect to.

/n
Suppress progress indicator.

/v
Verbose output for troubleshooting.

/p=Port
Connect to a server port other than the standard FTP port number of 21.

/f=ScriptFile
A file containing commands for the client to send to the server upon connection. Simple parameter substitution is performed, with the first element of *ScriptParameters* accessible as "%1," etc.

ScriptParameters
Parameters to pass into the *ScriptFile*.

Return Codes

0	Success
1	Parameter error
2	SOCKETS not loaded
3	User aborted
4	Transfer aborted
5	Error writing local file
6	Error reading local file
Other	Server returned error response code; to find that error code, add 390 to the response code returned by FTP. The result will always be greater than or equal to 400 in this case.

Example

```
FTP /n FTP.cdrom.com /f=getfile.scr /.1/idgames/idstuff/quake quakepix.zip

(The file GETFILE.SCR):
user anonymous
pass root@
cd %1
binary
get %2
quit
```

Example Output

```
230-Welcome to wcarchive - home FTP site of Walnut Creek CDROM.
230-There are currently 4252 users out of 5000 possible.
```

FTP Commands

The commands entered at the FTP client can be interpreted and translated to standard FTP commands to be sent to the server. The FTP server might recognise more, or less, commands than the standard list of commands as specified in RFC 959. The **site**

command is always server dependent. Some of the standard commands are implemented differently in various servers.

Useful things to note are:

1. The **put** and **get** commands allow multiple file transfers by usage of wild card characters. When **get**ting files with paths or long names, no translation of foreign file names are done. Specify a valid DOS ***local_file*** name.
2. A short directory list (NLST) is obtained by **ls** and the long list with **dir**.
3. Some of the commands can be abbreviated.
4. Some commands are aliases added for user comfort like **bye**, **exit** and **quit**; **get** and **mget**; and **put** and **mput**.
5. The optional **[*local_file*]** parameter will, when specified, cause the output of that command to be logged to a file. By specifying the file as PRN you can get immediate printouts.
6. On some servers you might specify the optional **[*remote_file*]** parameter as PRN or the printer output device to do remote printing. (See also the **site nopath** command for the SOCKETS FTP server.)
7. The **F3** key and **spacebar** can be used to recall the last command word by word.

Below is a list of commands recognised by the SOCKETS FTP client (some FTP servers might not offer all the facilities):

abort cancel an incomplete transfer
append "put" a file at the server but append it if the file exists
ascii synonym for **type a**
binary synonym for **type i**
bye synonym for **quit**
cd *directory* synonym for **cwd**
cwd *directory* change server directory
dele *file* delete a server file
dir [*file* | *directory* [*local_file*]] synonym for **list**
exit synonym for **quit**
get *remote_file(s)* [*local_file*] transfer a file from the server in the current mode (**type**)
image synonym for **type i**
ls [*file* | *directory* [*local_file*]] synonym for **nlst**
lcd *directory* perform a local change directory
ldir [*file* | *directory*] give a local directory listing
list [*file* | *directory* [*local_file*]] give a long directory listing
mget *remote_file(s)* [*local_file*] synonym for **get**
mkdir *remote_directory* create a server directory
mput *local_file(s)* [*remote_file*] synonym for **put**
nlst [*file* | *directory* [*local_file*]] give a short names-only directory listing
pass [*password*] password for username
pasv [on | off] report or change the status of the passive transfer mode to enable firewall friendly file transfers. (The SOCKETS FTP client always tries to switch passive mode on at the start of a session.)
put *local_file(s)* [*remote_file*] transfer a file to the server in the current mode (**type**)
pwd print working directory at server

quit terminate FTP session
quote *remote_command* [*args* ...] send a command to the server without any interpretation
rmdir *remote_directory* remove (delete) a server directory
rnfr *existing_filename* rename a file, command 1 of 2
rnto *new_filename* rename a file, command 2 of 2
site *sub-command* send server specific commands
size *file* report the file size in bytes as a 213 message
shell shell to DOS for IFTP.EXE
stat report the status of a transfer or active connections
system return operating system information from the server
type [i I a] report or select the file transfer mode: image (binary) or ASCII
user [*username*] username to logon
verbose [on | off] verbose mode reports more of the FTP negotiations

FTPAPI

FTPAPI is both a server and client for the FTP protocol that loads as a TSR. It should be called directly from your application. The documentation for this interface is found in FTPAPI.H. A complete "server and multiple client" sample program is provided as FTPTEST.C and FTPTEST.EXE.

Syntax

FTPAPI [options] [Port]

Options

/m=MemorySize

/c

/s

/u

Remarks

/m=MemorySize

The number of bytes of memory available to the server. This value defaults to 32768.

/c

Running **FTPAPI** again with this option disables the TSR's listen socket.

/s

Running **FTPAPI** again with this option displays the TSR's status.

/u

Running **FTPAPI** again with this option unloads the TSR.

[Port]

FTPAPI listens on the listed port. This parameter defaults to the standard FTP port number of 21.

Configuration File

FTPAPI uses the standard SOCKET.UPW file for validating logins. See **FTPD** for more information.

Example

```
FTPAPI /m=40000 42
```

Example Output

```
Datalight Sockets Resident FTP Services v7.10 (Revision 4.00.1145)
Copyright © 1989-2001 Datalight, Inc.
Portions Copyright © GPvNO 2001


Starting LISTEN on port 42
Going resident
```

Print Applications

These applications allow hosts to share printers. The LPD protocol is standard in the Unix community, and the SPRINT protocol is proprietary to Datalight SOCKETS.

LPR

LPR is a printer client for UNIX-style printer servers. There is no matching **LPD** server for SOCKETS.

Syntax

LPR /s=Server /p=Printer /u=Agent Filename

Options

/r

/q

/l=Port

/h=LocalHostName

/c=JobClass

/j=JobName

/n

/t

Remarks

/q

Query Mode. Can be followed by agent names or job numbers to filter output.

/r

Remove Mode. May be followed by job numbers to specify jobs to remove.

/s=Server
Hostname of the print server.

/p=Printer
Name of the printer device on the server to be used for output.

/u=Agent
User name on the server. Used for identification.

Filename
Local file name to spool to the server.

/l=Port
Connect to the specified port on the server rather than the standard port number of 515.

/h=LocalHostName
Name of the local host for job identification purposes.

/c=JobClass
Name of the job class for job identification and scheduling purposes.

/j=JobName
Name of the job, for identification purposes; defaults to the local file name if not specified.

/n
Run without user interaction.

/t
Text filter. Strips all unprintable characters before printing.

Example

```
LPR /n /s=10.0.0.1 /p=prn0 /u=Tester output.dat
LPR /n /s=10.0.0.1 /p=prn0 /u=Tester /q
LPR /n /s=10.0.0.1 /p=prn0 /u=Tester /r
```

Example Output

```
Printing to 10.0.0.1:515
```

SPRINT

SPRINT is a simple printer client that prints through a SOCKETS host configured as a print server. To configure a SOCKETS host as a print server, see the **start prntsrv** command in Chapter 6, SOCKETS Configuration Reference .

Syntax

SPRINT Server Filename [options]

Options

Port

Mask

Remarks

Server

Hostname of the print server.

Filename

Local file name to spool to the server.

Port

Connect to the specified port on the server rather than the standard port number of 10.

Mask

Used for status reporting, this parameter only works if the server was configured with a mask.

Example

```
SPRINT 10.0.0.1 output.dat
```

Example Output

```
Printing to 10.0.0.1:515
```

Chapter 8, SOCKETS API Reference

SOCKETS API Overview

This chapter describes the SOCKETS API, which 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 **GetSocket**() 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 **ReleaseSocket**(). 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 **stdout**. 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). Refer also to “Using STREAM and DATAGRAM Service” on page 102.
- 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 *dwRemoteHost*, 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 dwHost, int iLength);

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 iSocket**)**;

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 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 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 iMaxid, long *plIflags, long *plOflags**)**;

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 or both the NET_FLG_PEEK and NET_FLG_NON_BLOCKING flags are 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 wFlags);

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 iSocket, char *pcBuf, WORD wLen, NET_ADDR *psFrom, WORD wFlags);

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 iSocket, char *pcBuf, WORD wLen, WORD wFlags);

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 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 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 iSocket);

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 iSocket);

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 iSocket, DWORD dwTime, int (far *lpHandler)(), DWORD dwHint);

Options

iSocket
Socket handle for the connection.

dwTime
Timer delay in milliseconds.

lpHandler
Far address of alarm callback. See the description of **SetAsyncNotification**() 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 iSocket, int iEvent, int (far *lpHandler)(),DWORD dwHint);

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 iSocket, int iEvent, DWORD dwArg**)**;

SetAsyncNotification(iSocket, iEvent, AsyncNotificationHandler, (DWORD)MyHandler);

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

Chapter 9, 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

CAPI Functions Used

- ReleaseSocket
- iNetErrNo
- WriteSocket
- ReadSocket
- GetPeerAddress
- ListenSocket
- SetSocketOption
- GetSocket
- ConnectSocket
- ResolveName

Includes and Defines

These includes and defines are needed by later code pieces.

```
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <process.h>
#include "compiler.h"
#include "capi.h"
#include "CHAT.pt"

#define CHAT_PORT 5000
#define MAX_CONS 30
```

```
#define BUF_SIZE 200

static int rgiSocks[MAX_CONS];            //all the sockets (passive and active)
static char rgszNames[MAX_CONS][80];      //Names associated with connections
static NET_ADDR sNetAddr;                 //for general use

static char rgcKeyBuf[BUF_SIZE];          //key input buffer
static int iKeyCount = 0;                 //number of characters in above buffer
```

Utility Functions

The following section of code shows three useful utility functions. Brief comments before each function should provide explanations.

/*

Create a human understandable string from a SOCKETS error code.

Argument:

uErrCode - The SOCKETS error code.

Returns:

A pointer to the (static) string representation of the error.

*/

```
char *GetErrorString(unsigned uErrCode)
{
    static char rgcUnk[30];
    static char *rgszErrs[] =
    {
        "NoErr",
        "InUse",
        "DOSErr",
        "NoMem",
        "NotNetconn",
        "IllegalOp",
        "BadPkt",
        "NoHost",
        "CantOpen",
        "NetUnreachable",
        "HostUnreachable",
        "PortUnreachable",
        "PortUnreachable",
        "TimeOut",
        "HostUnknown",
        "NoServers",
        "ServerErr",
        "BadFormat",
        "BadArg",
        "EOF",
        "Reset",
        "WouldBlock",
        "UnBound",
        "NoDesc",
        "BadSysCall",
        "CantBroadcast",
```

```
            "NotEstab",
            "ReEntry",
        };

        if (uErrCode == ERR_API_NOT_LOADED)
            return "Sockets not loaded";
        if ((uErrCode & 0xff) > ERR_RE_ENTRY)
        {
            sprintf(rgcUnk,"Unknown error 0x%04X",uErrCode);
            return rgcUnk;
        }
        return rgszErrs[uErrCode & 0xff];
    }


/*

Show data on screen.

For now we just use vprintf to print the data and reprint any stuff that was being edited.

Accepts variable number of parameters, exactly like printf, and process them using the va_list/va_args method.

Arguments:

   pFormat and ellipsis - Exactly the same as for printf().

*/
    void Aprintf (char *pFormat, ...)
    {
        va_list pArgs;

        if (iKeyCount)//data been edited, start on newline
            printf("\n");

        va_start(pArgs, pFormat);
        vprintf(pFormat, pArgs);
        va_end(pArgs);

        //print the old edited stuff (if any)
        if (iKeyCount)
        {
            rgcKeyBuf[iKeyCount] = 0;
            printf("%s", rgcKeyBuf);
        }
    }


/*

Create a string representation of the IP address and port, in the form

   a.b.c.d:port, eg 196.10.180.3:1400.

Max length is 22 bytes.

Arguments:
```

psAddr - pointer to NET_ADDR structure containing address of host.

Returns:

Pointer to (static) array containing null-terminated string.

*/

```
static char *WriteName(NET_ADDR *psAddr)
{
    static char rgcName[22];

    sprintf(rgcName, "%u.%u.%u.%u:%u",
        ((BYTE *)&psAddr->dwRemoteHost)[0],
        ((BYTE *)&psAddr->dwRemoteHost)[1],
        ((BYTE *)&psAddr->dwRemoteHost)[2],
        ((BYTE *)&psAddr->dwRemoteHost)[3],
        psAddr->wRemotePort
    );
    return rgcName;
}
```

Array Maintenance Functions

The uses of these functions are not quite as obvious as the last set. They operate on an array of socket handles. If an array element is zero, then it is not allocated. Otherwise, it is an open socket that may have data pending for send or receive. One useful function is to find the first unallocated entry so that it can be allocated. Another would be to find the next allocated entry to test its data-pending state. The following functions implement all of the code needed to perform these routines.

/*

Gets the first not-assigned entry in the rgiSocks array.

Returns:

The first nonzero index or -1 if all in use.

*/

```
static int GetFirstOpen(void)
{
    int iRet = 0;
    while (iRet < MAX_CONS)
    {
        if (rgiSocks[iRet] == 0)
            return iRet;
        iRet++;
    }
    return -1;
}
```

/*

The next two functions implements an enumerator. To initialize (or reset),

call resetEnum. Successive calls to getNextIndex returns all the used

indexes. When there is no more used indexes, -1 is returned

*/

```
  static int iPrev = -1;     //privately used by next two functions
  static void ResetEnum(void)
  {
      iPrev = -1;
  }
```

/*

Gets the next not 0 entry in the rgiSocks table.

Returns:

The index of the next entry or -1 if no more.

*/

```
  static int GetNextIndex(void)
  {
      while (++iPrev < MAX_CONS)
      {
          if (rgiSocks[iPrev] != 0)
              return iPrev;
      }
      return -1;
  }
```

Connection Functions

These functions are simple wrappers that isolate common, redundant code for easier debugging and use. They are so common that they can be copied verbatim into your application if you like.

/*

Open a TCP connection to the specified host and the default CHAT port.

Argument:

pHostName - A pointer to the name of the host we want to connect to

Returns:

A descriptor of the newly created socket.

*/

```
  int ConnectTo(char* pcHostName)
  {
      int iSock;      //descriptor

      memset(&sNetAddr, 0, sizeof(NET_ADDR));

      sNetAddr.wRemotePort = CHAT_PORT;
```

```
    // execute several commands and test for error at each step
    if ((sNetAddr.dwRemoteHost = ResolveName(pcHostName, 0, 0)) == 0)
    {
        Aprintf("Error on ResolveName: %s\n", GetErrorString(SocketsErrNo));
    }
    else if ((iSock = GetSocket()) < 0)
    {
        Aprintf("Error on GetSocket(): %s\n", GetErrorString(SocketsErrNo));
    }
    else if (SetSocketOption(iSock, 0, NET_OPT_NON_BLOCKING, 1, 1) < 0)
    {
        Aprintf("Error on SetSocketOption(): %s\n", GetErrorString(SocketsErrNo));
    }
    else if (ConnectSocket(iSock, STREAM, &sNetAddr) < 0)
    {
        Aprintf("Error on ConnectSocket: %s\n", GetErrorString(SocketsErrNo));
    }
    else
    {
        Aprintf("Trying to connect to %s (%s)\n",pcHostName, WriteName(&sNetAddr));
        return iSock;
    }

    Aprintf("Error on connect to %s - no con\n", pcHostName);
    return 0;
}
```

/*

Start a TCP server on specified port. (Returns immediately)

Argument:

iPort - The TCP port number to listen on.

Returns:

A descriptor of the server socket, or 0 if an error occured.

*/

```
int StartListen(int iPort)
{
    int iSock;

    memset(&sNetAddr, 0, sizeof(NET_ADDR));
    sNetAddr.wLocalPort = iPort;

    if ((iSock = GetSocket()) < 0)
    {
        Aprintf("Error on serv GetSocket(): %s\n", GetErrorString(SocketsErrNo));
    }
    else if (SetSocketOption(iSock, 0, NET_OPT_NON_BLOCKING, 1, 1) < 0)
    {
        Aprintf("Error on serv setOpt(): %s\n", GetErrorString(SocketsErrNo));
    }
    else if (ListenSocket(iSock, STREAM, &sNetAddr) < 0)
    {
        Aprintf("Error on serv net_listen: %s\n", GetErrorString(SocketsErrNo));
    }
    else
```

```
    {
        return iSock;
    }
    return 0;
}
```

CHAT Code Loop

The following is the main code loop for the CHAT program. It relies on all the functions above, and builds on top of them using a common socket-polling methodology.

/*

Loop and look for both user input or network input.

*/

```
void main(void)
{
    //listening socket, copied to rgiSocks once connection is made
    int iListenSock;

    //general variables
    char rgcBuf[BUF_SIZE];
    int iIndex, iLength;
    char cCh;

    // clear all client sockets
    for (iIndex = 0; iIndex < MAX_CONS; iIndex++)
        rgiSocks[iIndex] = 0;

    // start the server
    iListenSock = StartListen(CHAT_PORT);

    // give a visual cue for users
    Aprintf("Press Alt-H for help\n");

    // loop forever, looking for network/keyboard input
    while (1)
    {
        //server part - see if new connection was opened
        iLength = ReadSocket(iListenSock, 0, 0, 0, 0);
        if (iLength == 0 || SocketsErrNo == ERR_WOULD_BLOCK)
        {
            // client is connected, allocate array entry
            if ((iIndex = GetFirstOpen()) != -1)
            {
                // record this as a client connection
                rgiSocks[iIndex] = iListenSock;

                // show the connection
                GetPeerAddress(iListenSock, &sNetAddr);
                sprintf( rgszNames[iIndex], "%s", WriteName(&sNetAddr));
                Aprintf("Connection made by %s\n", rgszNames[iIndex]);

                // hello the new client
                iLength = WriteSocket(rgiSocks[iIndex], "(server)Hello!", 14, 0);
                if (iLength < 0)
                {
                    Aprintf("Error on hello: %s\n", GetErrorString(SocketsErrNo));
```

```
        }
    }
    else
    {
        // no free slots available
        ReleaseSocket(iListenSock);
    }

    // start another server socket
    iListenSock = StartListen(CHAT_PORT);
}
else if (SocketsErrNo != ERR_NOT_ESTAB)
{
    // After a normal read, NOT_ESTAB is the
    // only valid error code.  Otherwise, a
    // serious error, tell the user and exit
    Aprintf("Error on server: %s. Exiting\n", GetErrorString(SocketsErrNo));
    goto exit;
}

//client part - scan for data ready to recieve
ResetEnum();
while ((iIndex = GetNextIndex()) != -1)
{
    // any data on this socket?
    iLength = ReadSocket(rgiSocks[iIndex], rgcBuf, BUF_SIZE, 0, 0);
    if (iLength <= 0)
    {
             if (SocketsErrNo == ERR_WOULD_BLOCK || SocketsErrNo ==
             ERR_NOT_ESTAB)
        {
            // normal socket, nothing to do
            continue;
        }

        if (iLength == 0 && SocketsErrNo == 0)
        {
            // client went away
            Aprintf("Peer (%s) has closed connection\n", rgszNames[iIndex]);
        }
         else
        {
            // error on the line
            Aprintf("Error on ReadSocket from %s:
            %s - closing\n", rgszNames[iIndex],
             GetErrorString(SocketsErrNo));
        }

        // dead client, free socket
        ReleaseSocket(rgiSocks[iIndex]);
        rgiSocks[iIndex] = 0;
    }
    else
    {
        // data received from client
        // force string termination
        rgcBuf[iLength] = 0;

        // show the node and string
        Aprintf("%s: %s\n", rgszNames[iIndex], rgcBuf);
    }
}
```

```
// local keyboard, were any keys hit?
if (kbhit())
{
    // yes.  which ASCII value?
    switch(cCh = getche())
    {
        // NULL -> extended key.
        case 0:
        {
            printf("\n");
            switch (cCh = getch())
            {
                // unexpected value?
                default:
                printf("Undefined function key: %d\n", cCh);
                //fall through

                // Alt+H - Help
                case 35:
                printf("Alt-C    Close connection\n"
                    "Alt-N    New connection\n"
                    "Alt-H    Help\n"
                    "Alt-L    List connections\n"
                    "Alt-X    eXit\n");
                break;

                // Alt+X - Exit
                case 45:
                goto exit;

                // Alt+N - Connect as client
                case 49:
                //make a new connection
                if ((iIndex = GetFirstOpen()) != -1)
                {
                    // get destination host
                    printf("Enter destination:");
                    gets(rgcBuf);

                    // attempt connection
                    rgiSocks[iIndex] = ConnectTo(rgcBuf);

                    // if success, copy the name
                    if ((rgiSocks[iIndex]) != 0)
                    {
                        strcpy( rgszNames[iIndex], rgcBuf );
                    }
                }
                else
                {
                    printf("Max number of connections in use\n");
                }
                break;

                // Alt+L - List connections
                case 38:
                printf("List of all connections\n");
                ResetEnum();
                while((iIndex=GetNextIndex()) != -1)
                {
                    printf("Connection #%d descriptor:%u name %s \n",
```

```
                iIndex, rgiSocks[iIndex], rgszNames[iIndex]);
    }
    printf("List end\n");
    break;

    // Alt+C - Close a connection
    case 46:
    printf("Enter connection to close:");
    iIndex = atoi(gets(rgcBuf));
    if (iIndex < 0 || iIndex > MAX_CONS)
    {
        printf("OUT of range:%d\n", iIndex);
    }
    else if (rgiSocks[iIndex])
    {
        ReleaseSocket(rgiSocks[iIndex] );
        rgiSocks[iIndex] = 0;
        printf("Closed %s\n", rgszNames[iIndex]);
    }
    else
    {
        printf("Connection %d not open\n", iIndex);
    }
    break;
}

// just for looks
Aprintf("");
break;

// normal key
default:
if (iKeyCount < BUF_SIZE)
{
    // we have room, store it
    rgcKeyBuf[iKeyCount++] = cCh;
    break;
}
// buffer full, force send now
// fall through

// enter - send buffer now
case '\r':
printf ("\n");
if (iKeyCount == 0)
   break;

//write data to all connections
ResetEnum();
while ((iIndex = GetNextIndex()) != -1)
{
    iLength = WriteSocket( rgiSocks[iIndex], rgcKeyBuf,
          iKeyCount, 0);
    if (iLength < 0)
    {
        // write failed!
        Aprintf("Error on NetWrite from %d %d bytes: %s -
        closing connection\n", iIndex, iKeyCount,
        GetErrorString(SocketsErrNo));
        ReleaseSocket( rgiSocks[iIndex]);
        rgiSocks[iIndex] = 0;
     }
```

```
                    }

                    // forget everything we just wrote
                    iKeyCount = 0;
                    break;
                }
            }
        }
    }

    // done running!
    exit:

    //release all sockets
    ResetEnum();
    while ((iIndex = GetNextIndex()) != -1)
        ReleaseSocket(rgiSocks[iIndex]);
    ReleaseSocket(iListenSock);
}
```

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.

CAPI Calls Used

- iNetErrNo
- ListenSocket
- SetSocketOption
- GetSocket
- ConnectSocket
- WriteSocket
- ReleaseSocket
- GetPeerAddress
- ReadSocket

Includes and Defines

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <process.h>
#include "compiler.h"
#include "capi.h"
#include "udpCHAT.pt"

#define CHAT_PORT 5000
#define BUF_SIZE 200
```

/*

It is not necessary to define NAME_LEN or MAX_CONS as we did with a stream connection.

We also do not need the static declarations that go along with MAX_CONS, rgiSocks and rgszNames.

*/

```
static char rgcKeyBuf[BUF_SIZE];    //for editing
static int iKeyCount = 0;           //number of characters in keybuf
static NET_ADDR sNetAddr;           //general use
```

Utility Functions

No changes are necessary within the "utility" functions between CHAT and udpCHAT. The error handling, print to screen, and address resolution are the same.

/*

Creates a human understandable string from a SOCKETS error code

Argument:

uErrCode - The SOCKETS error code

Returns:

A pointer to the (static) string representation of the error

*/

```
char *Err(unsigned uErrCode)
{
      static char rgcUnk[30];
      static char *rgszErrs[] =
      {
            "NoErr",
            "InUse",
            "DOSErr",
            "NoMem",
```

```
                "NotNetconn",
                "IllegalOp",
                "BadPkt",
                "NoHost",
                "CantOpen",
                "NetUnreachable",
                "HostUnreachable",
                "ProtUnreachable",
                "PortUnreachable",
                "TimeOut",
                "HostUnknown",
                "NoServers",
                "ServerErr",
                "BadFormat",
                "BadArg",
                "EOF",
                "Reset",
                "WouldBlock",
                "UnBound",
                "NoDesc",
                "BadSysCall",
                "CantBroadcast",
                "NotEstab",
                "ReEntry",
        };

        if (uErrCode == ERR_API_NOT_LOADED)
                return "Sockets API not loaded";
        if ((uErrCode & 0xff) > ERR_RE_ENTRY)
        {
                sprintf(rgcUnk,"Unknown error 0x%04X",uErrCode);
                return rgcUnk;
        }
        return rgszErrs[uErrCode & 0xff];
  }


/*
```

Show data on screen.

For now we just use vprintf to print the data and reprint any thing being edited.

Arguments:

Format and eclipse - Exactly the same as for prinf().

```
*/
  void Aprintf (char *pFormat, ...)
  {
        va_list pArgs;

        if (iKeyCount)//data been edited, start on newline
                printf("\n");

        va_start(pArgs, pFormat);
        vprintf(pFormat, pArgs);
        va_end(pArgs);

        //print the old edited stuff (if any)
```

```
        if (iKeyCount)
        {
                rgcKeyBuf[iKeyCount] = 0;
                printf("%s", rgcKeyBuf);
        }
  }
```

/*

Create a string representation of the IP address and port, in the form

a.b.c.d:port, eg 196.10.180.3:1400.

Max length is 22 bytes.

Arguments:

psAddr - pointer to NET_ADDR structure containing address of host.

Returns:

Pointer to (static) array containing null-terminated string.

*/

```
  static char *WriteName(NET_ADDR *psAddr)
  {
        static char rgcName[22];

        sprintf(rgcName, "%u.%u.%u.%u:%u",
                ((BYTE *)&psAddr->dwRemoteHost)[0],
                ((BYTE *)&psAddr->dwRemoteHost)[1],
                ((BYTE *)&psAddr->dwRemoteHost)[2],
                ((BYTE *)&psAddr->dwRemoteHost)[3],
                psAddr->wRemotePort
        );
        return rgcName;
  }
```

Array Maintenance Functions

Removed in UDP CHAT.

Connection Functions

/*

Sits in a loop and check for two things:

1. If any data was received from the network

   If data was received, it is displayed on screen

2. If the user has entered any data

   If the character entered was a newline, the previously entered data is

send, else the character is stored in a buffer.

```
*/
  void main(void)
  {
        char rgcMyNameBuf[NAME_LEN];//buffer containing my (arbritary) name
        char *pMyName;          //pointer to it
        int iMaxInput;          //maximum no of bytes before we must send (to keep all
                                //in buffer)
        char rgcBuf[BUF_SIZE];//general buffer, used for sending and receiving
        int iRcvSock;           //socket for receiving packets
        int iSendSock;          //socket for sending packets
        int iLength;            //Length of string read or written
        char cCh;               //Character read
        char cMoreInfo;         //Whether or not to display additional information

        printf("Sockets UDP CHAT client\n");
        printf("Copyright (C) 1999 Datalight, Inc.\nAll Rights Reserved\n\n");

        //get user name and set variables accordingly
        printf("Enter your name (send with all your messages):");
        rgcMyNameBuf[0] = NAME_LEN - 2;
        pMyName = cgets(rgcMyNameBuf);
        iMaxInput = BUF_SIZE - rgcMyNameBuf[1] - 10;

        //get more info option
        printf("\nDo you want to see the ip addresses of senders? [Y/N]");
        rgcBuf[0] = 2;
        cgets(rgcBuf);
        cMoreInfo = (rgcBuf[2] == 'Y' ? 1 : 0);

        printf("\nPress Alt-X to exit\n");

        //start to listen for datagrams
        iRcvSock = StartListen(CHAT_PORT);
        iSendSock = GetClientSock();
        if (iRcvSock == 0 || iSendSock == 0)
                return;
        //tell the world I am on the air
        iLength = sprintf(rgcBuf, "%s came on the air", pMyName);
        if (WriteSocket(iSendSock, rgcBuf, iLength, NET_FLG_BROADCAST) < 0)//error
                Aprintf("Error on Sending %d bytes: %s\n",iLength, Err(iNetErrNo));

        while (1)
        {
                //server part - see if we received data
                iLength = ReadSocket(iRcvSock, rgcBuf, BUF_SIZE, 0, 0);
                if (iNetErrNo != 0 && iNetErrNo != ERR_WOULD_BLOCK)
                        Aprintf("Error on Netread: %s\n", Err(iNetErrNo));
                if (iLength > 0)
                {
                        rgcBuf[iLength] = 0;
                        //no color now l = sNetAddr.dwRemoteHost & 0xff;
                        //some color identifing host
                        if (cMoreInfo)//give the senders ip address as well
                        {
                                GetPeerAddress(iRcvSock, &sNetAddr);
                                Aprintf("%s (from %s)\n", rgcBuf, WriteName(&sNetAddr));
                        }
                        else
                                Aprintf("%s\n");
```

```
                }
                if (kbhit())
                {
                        switch(cCh = getche())
                        {
                        case 0:                         /* function key */
                                if (getch() == 45)//Alt-X  exit
                                {
                                        ReleaseSocket(iRcvSock);
                                        ReleaseSocket(iSendSock);
                                        return;
                                }
                                break;
                        default:
                                if (iKeyCount < iMaxInput)
                                {
                                        rgcKeyBuf[iKeyCount++] = cCh;
                                        break;
                                }
                                //fall thru
                        case '\r':
                        //case '\n':
                                putch('\n');
                                if (iKeyCount == 0)//nothing to do
                                        break;
                                //everyone should identify himself
                                rgcKeyBuf[iKeyCount] = 0;
                                iLength = sprintf(rgcBuf, "%s :%s", pMyName, rgcKeyBuf);
                                //broadcast data
                                if (WriteSocket(iSendSock, rgcBuf, iLength,
                                NET_FLG_BROADCAST) < 0)//error
                                        Aprintf("Error on NetWrite %d bytes: %s\n",
                                        iLength, Err(iNetErrNo));
                                iKeyCount = 0;
                                break;
                        }
                }
        }
  }
```

/*

Get a socket descriptor that can be used to send UDP packets

Returns:

A descriptor of the socket, or 0 if an error occured. (Returns immediately)

*/

```
  static int GetClientSock(void)
  {
        int iSock;

        memset(&sNetAddr, 0, sizeof(NET_ADDR));
        sNetAddr.wRemotePort = CHAT_PORT;

        if ((iSock = GetSocket()) < 0)
                Aprintf("Error on GetSocket(): %s\n",Err(iNetErrNo));
        else if (SetSocketOption(iSock, 0, NET_OPT_NON_BLOCKING, 1, 1) < 0)
                Aprintf("Error on setOpt(): %s\n",Err(iNetErrNo));
```

```
        else if (ConnectSocket(iSock, DATA_GRAM, &sNetAddr) < 0)
                Aprintf("Error on net_connect: %s\n",Err(iNetErrNo));
        else
        {
                Aprintf("Client sock successfully created\n");
                return iSock;
        }

        return 0;
  }
/*
```

Start a UDP server on specified port.

Returns:

A descriptor of the server socket, or 0 if an error occured.

(Returns immediately)

```
*/
  int StartListen(int iPort)
  {
        int iServSock; //the new socket

        memset(&sNetAddr, 0, sizeof(NET_ADDR));
        sNetAddr.wLocalPort = iPort;

        if ((iServSock = GetSocket()) < 0)
                Aprintf("Error on serv GetSocket(): %s\n",Err(iNetErrNo));
        else if (SetSocketOption(iServSock, 0, NET_OPT_NON_BLOCKING, 1, 1) < 0)
                Aprintf("Error on serv setOpt(): %s\n", Err(iNetErrNo));
        else if (ListenSocket(iServSock, DATA_GRAM, &sNetAddr) < 0)
                Aprintf("Error on serv net_listen: %s\n", Err(iNetErrNo));
        else
        {
                Aprintf("Server sock successfully created\n");
                return iServSock;
        }
        return 0;
  }
```

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”.

CAPI Calls Used

- iNetErrNo
- WriteSocket
- JoinGroup
- LeaveGroup
- ReleaseSocket
- ReadSocket
- ConnectSocket
- SetSocketOption
- GetSocket
- ResolveName

Includes and Defines

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#include <process.h>
#include "compiler.h"
#include "capi.h"
#include "udpCHAT.pt"

#define CHAT_PORT 5000
#define BUF_SIZE 200
#define NAME_LEN     20                    //my name

static char rgcKeyBuf[BUF_SIZE];           //for editing
static int iKeyCount = 0;                  //number of characters in keybuf
static NET_ADDR sNetAddr;                  //general use
```

Utility Functions

/*

Create a human understandable string from a SOCKETS error code

Argument:

uErrCode - The SOCKETS error code

Returns:

A pointer to the (static) string representation of the error

*/

```
  char *Err(unsigned uErrCode)
  {
        static char rgcUnk[30];
        static char *rgszErrs[] =
        {
                "NoErr",
                "InUse",
                "DOSErr",
                "NoMem",
                "NotNetconn",
                "IllegalOp",
                "BadPkt",
                "NoHost",
                "CantOpen",
                "NetUnreachable",
                "HostUnreachable",
                "ProtUnreachable",
                "PortUnreachable",
                "TimeOut",
                "HostUnknown",
                "NoServers",
                "ServerErr",
                "BadFormat",
                "BadArg",
                "EOF",
                "Reset",
                "WouldBlock",
                "UnBound",
                "NoDesc",
                "BadSysCall",
                "CantBroadcast",
                "NotEstab",
                "ReEntry",
        };

        if (uErrCode == ERR_API_NOT_LOADED)
                return "Sockets API not loaded";
        if ((uErrCode & 0xff) > ERR_RE_ENTRY)
        {
                sprintf(rgcUnk,"Unknown error 0x%04X",uErrCode);
                return rgcUnk;
        }
        return rgszErrs[uErrCode & 0xff];
  }
```

/*

Show data on screen.

For now we just use vprintf to print the data and reprint any stuff that was being edited.

Arguments:

Format and eclipse - Exactly the same as for prinf().

```
*/
  void Aprintf (char *pFormat, ...)
  {
        va_list pArgs;

        if (iKeyCount) //data been edited, start on newline
                printf("\n");

        va_start(pArgs, pFormat);
        vprintf(pFormat, pArgs);
        va_end(pArgs);

        //print the old edited stuff (if any)
        if (iKeyCount)
        {
                rgcKeyBuf[iKeyCount] = 0;
                printf("%s", rgcKeyBuf);
        }
  }
```

/*

Create a string representation of the IP address and port, in the form

a.b.c.d:port, eg 196.10.180.3:1400.

Max length is 22 bytes.

Arguments:

psAddr - pointer to NET_ADDR structure containing address of host.

Returns:

Pointer to (static) array containing null-terminated string.

```
*/
  static char *WriteName(NET_ADDR *psAddr)
  {
        static char rgcName[22];

        sprintf(rgcName, "%u.%u.%u.%u:%u",
                ((BYTE *)&psAddr->dwRemoteHost)[0],
                ((BYTE *)&psAddr->dwRemoteHost)[1],
                ((BYTE *)&psAddr->dwRemoteHost)[2],
                ((BYTE *)&psAddr->dwRemoteHost)[3],
                psAddr->wRemotePort
        );
        return rgcName;
  }
```

Connection Functions

/*

Sit in a loop and check for two things:

1. If any data was received from the network

   If data was received, it is displayed on screen

2. If the user has entered any data

   If the character entered was a newline, the previously entered data is

   sent, else the character is stored in a buffer.

*/

```
  void main(void)
  {
        char rgcMyNameBuf[NAME_LEN];  //buffer containing my (arbitrary) name
        char *pMyName;                //pointer to it
        int iMaxInput;                //maximum no of bytes before we must send
                                      //(to keep all in buffer)
        char rgcBuf[BUF_SIZE];        //general buffer, used for sending and receiving
        int iSock;                    //socket for receiving and sending packets
        int iLength;                  //Length of string read or written
        char cCh;                     //Character read
        char cMoreInfo;               //Whether or not to display additional
  information
```

/*

add the following to main

*/

```
        WORD wNoMCEcho;               // Suppress display of own messages
        DWORD dwGroupAddress;         // Group IP address to use

        printf("Sockets UDP CHAT client\n");
        printf("Copyright (C) 1999 Datalight, Inc.\nAll Rights Reserved\n\n");

        //get user name and set variables accordingly
        printf("Enter your name (send with all your messages):");
        rgcMyNameBuf[0] = NAME_LEN - 2;
        pMyName = cgets(rgcMyNameBuf);
        iMaxInput = BUF_SIZE - rgcMyNameBuf[1] - 10;

        //get more info option
        printf("\nDo you want to see the ip addresses of senders? [Y]/N");
        rgcBuf[0] = 2;
        cgets(rgcBuf);
        cMoreInfo = (toupper(rgcBuf[2]) == 'N' ? 0 : 1);
        printf("\nDo you want to suppress your own messages? [Y]/N");
        rgcBuf[0] = 2;
        cgets(rgcBuf);
        wNoMCEcho = (toupper(rgcBuf[2]) == 'N' ? 0 : NET_FLG_MC_NOECHO);


        printf("\nPress Alt-X to exit\n");

        if ((dwGroupAddress = ResolveName("229.1.2.3", rgcBuf, BUF_SIZE)) == 0)
               Aprintf("Error on ResolveName(): %s\n", Err(iNetErrNo));
```

```
// tell IP that we want to receive multicast datagrams
// on the default interface
if (JoinGroup(dwGroupAddress,0) < 0)
        Aprintf("Error on JoinGroup(): %s\n", Err(iNetErrNo));

sNetAddr.dwRemoteHost = dwGroupAddress;
sNetAddr.wRemotePort = CHAT_PORT;
sNetAddr.wLocalPort = CHAT_PORT;

if ((iSock = GetSocket()) < 0)
{
        Aprintf("Error on GetSocket(): %s\n",Err(iNetErrNo));
        return;
}
if (SetSocketOption(iSock, 0, NET_OPT_NON_BLOCKING, 1, 1) < 0)
{
        Aprintf("Error on SetSocketOpt(): %s\n",Err(iNetErrNo));
        return;
}
if (ConnectSocket(iSock, DATA_GRAM, &sNetAddr) < 0)
{
        Aprintf("Error on ConnectSocket(): %s\n",Err(iNetErrNo));
        return;
}
Aprintf("Socket successfully created\n");

// tell the world I'm on the air
iLength = sprintf(rgcBuf, "%s came on the air", pMyName);
if (WriteSocket(iSock, rgcBuf, iLength, wNoMCEcho) < 0)//error
        Aprintf("Error on Sending %d bytes: %s\n",iLength, Err(iNetErrNo));

while (1)
{
        // see if we received data
        sNetAddr.dwRemoteHost = 0l;
        iLength = ReadSocket(iSock, rgcBuf, BUF_SIZE, &sNetAddr, 0);
        if (iNetErrNo != 0 && iNetErrNo != ERR_WOULD_BLOCK)
                Aprintf("Error on Netread: %s\n", Err(iNetErrNo));
        if (iLength > 0)
        {
                rgcBuf[iLength] = 0;
                //no color now l = sNetAddr.dwRemoteHost & 0xff;
                //some color identifing host
                if (cMoreInfo) //give the sender's ip address as well
                        Aprintf("From %s - ", WriteName(&sNetAddr));
                Aprintf("%s\n",rgcBuf);
        }
        if (kbhit())
        {
                switch(cCh = getche())
                {
                case 0:                                 // function key
                        if ((cCh = getch()) == 45)      // Alt-X  exit
                        {
                                ReleaseSocket(iSock);
                                LeaveGroup(dwGroupAddress,0);
                                return;
                        }
                        if (cCh == 16)                  // Alt-Q generate query
                        {
                                JoinGroup(0,0);
                        }
```

```
                                        break;
                                default:
                                        if (iKeyCount < iMaxInput)
                                        {
                                                rgcKeyBuf[iKeyCount++] = cCh;
                                                break;
                                        }
                                        //fall thru
                                case '\r':
                                //case '\n':
                                        putch('\n');
                                        if (iKeyCount == 0)    //nothing to do
                                                break;
                                        //everyone should identify himself
                                        rgcKeyBuf[iKeyCount] = 0;
                                        iLength = sprintf(rgcBuf, "%s: %s", pMyName, rgcKeyBuf);
                                        //broadcast data
                                        if (WriteSocket(iSock, rgcBuf, iLength, wNoMCEcho) <
                                        0)//error
                                                Aprintf("Error on NetWrite %d bytes: %s\n",
                                                iLength, Err(iNetErrNo));
                                        iKeyCount = 0;
                                        break;
                                }
                        }
                }
        }
```

Chapter 10, SOCKETS Server Reference

Topics within This Chapter:

Overview

The SOCKETS web server, HTTPD, is a small, fast, reliable and extendable web server. Apart from the minimum required file download capability, the following additional capabilities are provided:

1. Remote Console - ability to gain terminal-type access to the server system, using a standard browser, without the need to install any software on the browser computer
2. Authentication – Both system wide and directory wise
3. CGI Extendibility – The ability to extend the server to create dynamic web pages, perform specialized tasks, etc.
4. A Server Side Includes (SSI) interface is provided using the CGI interface, enabling a user to create web pages using HTML templates with variable names, which is substituted in-time with specific values
5. Ability to run as a background process
6. Flexibility to control physical parameters such as memory usage and number of connections

Server

The server runs as a single DOS TSR using SOCKETS as the TCP/IP stack and providing two distinct services, i.e. a Web (HTTP) service and a Remote Console service. Note that the Server processor is the (ROM) DOS embedded system.

The HTTP server is used to send static web pages existing as files on the server to a remote client (browser). Dynamic pages can be generated in two ways:

1. By calling an external CGI handler, the server provides an API to external handlers. A Server Side Includes (SSI) interface is provided as well, which makes it very easy to create powerful interactive web pages.

2. By spawning programs with a relatively short execution time to generate the pages through a mechanism similar to CGI, the basic mechanism used by CGI is that arbitrary programs can be spawned from the web server with input as received from the remote browser and output that can be sent to the browser.

The Remote Console server accepts input from a remote client that is fed to the keyboard buffer for use by an arbitrary program using it. It also monitors the screen display buffer area and sends screen information to the remote client.

The socket password file controls authentication. Authentication is user specific and may also differ from directory to directory. It may also be put off for either some or all users. See the section on authentication.

The HTTP server can support multiple simultaneous sessions. The GET and POST request methods are implemented as well as the following MIME types:

text/html, text/plain, image/gif, image/jpeg, image/jpeg and application/octet-stream.

The MIME type is determined by the file extension.

Remote Console

Initialization

The client (browser) will initialize a remote session. An HTTP connection will be made to the HTTP server. The downloaded page will contain the applet that will automatically connect to the RCS on TCP port 81. An example download page is supplied as REMCON.HTM.

Almost any application e.g. a text editor can be run on the server. The remote keyboard and display control the application as if they were locally attached.

On the remote side, the Java Applet acts as a simple terminal emulator that displays what it receives from the server and sends what is entered from the keyboard to the server.

It is not required to have a real display adapter on the embedded system server, only to have display buffer memory.

When RCCLI.EXE is used, a connection to port 81 on the server is established.

Remote Console Server (RCS)

When a new connection is made, all the screen data, as well as the cursor position, is sent to the client. Subsequently the RCS keeps a watch on the video memory and cursor position and whenever a change is detected, the RCS sends the changed data to the Java applet.

Keyboard data received from the client is passed to the keyboard buffer making it available as keyboard input for use by any application executing on the server.

Remote Console Client

The remote console client exists as a Java 1.1 applet, supplied in RC.JAR, on any Java 1.1 compliant browser. Please note that a security certificate has not been compiled into RC.JAR so it is not compliant with versions of the Netscape browser that require a security certificate to run Java applets. A DOS based client using SOCKETS is also supplied as RCCLI.EXE

Java Applet

Receives and displays screen data from RCS. Key presses are translated and sent to RCS.

RCCLI.exe

Run from DOS command line: RCCLI <host> <port>.

CGI

HTTPD Extension CGI

The HTTPD Extension CGI works as follows: The extension has to implement one function called the *callback function*. The server has a number of functions that the extension may use, e.g. *HttpSendData*. They are designed to give the extension sufficient control over any http request.

Detail

The SOCKETS HTTP server (HTTPD) provides a facility to call functions in other modules which may be TSR or transient programs. These functions are referred to as "HTTPD extensions". HTTPD must be loaded as a TSR using the /r switch. It provides an API via software interrupts that defaults to Int 63h. The API can be located by searching for a signature containing SockHTTPD starting 10 bytes before the interrupt entry point and terminated by a 0 byte.

A **CGI adapter** is provided that simplifies the communication with the server. It is located in a file called '*cgiadap.c'*. When using the cgi adapter, the adapter finds the signature and performs interrupts. It also intercepts the callback function and performs a stack and context switch, which makes implementing an extension much easier. See the sample extensions.

An HTTPD extension registers interest in a specific URL by calling the HttpRegister() API specifying a "path". Note that this path has nothing to do with an actual file path on the server and will override any real path that may be used for serving static pages. The HttpRegister() function also specifies a Callback function to be called when the actual request is received by HTTPD, a DWORD user id to be used in callbacks and whether requests should be allowed to overlap, i.e. a new request can be received while still servicing a previous request or requests.

The Callback function will be called when a request for the registered path is received and as many times afterwards as is necessary to complete the request. It is called with a parameter structure specifying the reason for the request, the user ID, an HTTPD handle and values specific to the reason for the callback, e.g. a pointer to the command line on the initial callback. Other reasons for calling the Callback function are to notify of new received data, connection closure by the peer, readiness to accept more data and connection errors. The callback must return a value to indicate that it is still busy handling the request, has completed the request or wants to abort the request with an error. The HTTPD handle will be constant and unique from the first callback to the completion of the request.

While in the Callback function, data can be read from the peer or sent to the peer and a file can be submitted to be sent to the peer.

Note: Extensions are responsible for sending all HTTP header fields to clients.

The following extensions have been developed for functional and demonstrational purposes.

SSI Interface

If you want to display the current date and time, or a certain CGI environment variable in your otherwise static document, you can go through the trouble of writing a CGI program that outputs this small amount of virtual data. Or better yet, you can use a powerful feature called Server Side Includes (or SSI).

Server Side Includes are directives which you can place into your HTML documents to output such data as environment variables and file statistics.

For a detailed introduction, please visit http://www.ora.com/info/cgi/ch05.html

A simple yet powerful interface is provided to perform Server Side Includes (SSI) tasks. A user only has to implement one predefined function and make use of only four API functions to unlock the power of SSI.

The working of the interface is described at the top of the header file *ssi.h*.

To use, include *ssicgi.c* in your project and include *ssi.h* in your source files. Take a look at *ssi.c* for a simple example.

CGI Examples

Five very simple examples are included to demonstrate the usage of the API. Source code is included, as well as Borland 3.1 project files.

Put all *.htm* and *.exe* files in the %HTTP_DIR% directory and start *HTTPD*. Load all the cgi programs (you may use *cgi.bat*). All is in place now and the examples may be accessed through *index.htm*.

The first four examples may operate in one of two modes:

As a TSR (resident) program: this is the default behavior. At this stage unloading of the TSR is not supported. De-registration is possible by loading the program again. This routine may be repeated.

As a transient program: use '/t' command line switch to activate. This option will immediately spawn 'command.com'. From this prompt other cgi programs may be loaded. The program exits when 'command.com' is exited by typing 'exit' at the prompt.

These programs are:

1. ***cgiecho*** A very simple program that accepts data from a user and echoes it back nicely formatted. Get *echoform.htm* from the browser.

2. ***cgicount*** A page visit counter. Only updates between sessions if transient (cgicount /t) Get *num.htm* from the browser.

3. ***cgiform*** Does the same as the old 'fill out the form and submit' utility. Get *caform.htm* from the browser.

4. ***SSI*** A very simple SSI implementation that demonstrates the SSI interfaces. *Template.htm* is filled by some variables. Get *ssi.htm* from the browser.

The fifth example, ***FFUR***, (Form-base File Upload Receiver) is only a transient program, but can easily be adapted to be similar to the rest. It handles the upload of a file as a POST command by filling out *ffur.htm.*

Passive Mode

The server may be run in passive mode by specifying a '/p' command line switch. When passive, the server will record network events but only handle them once it is triggered by a CGI user. The user does this by calling HTTPGETStatus(), which only returns the number of active connections. However, this function gives the server a chance to perform all the normal server routines.

Server Memory

The server's memory usage may be controlled in two ways:

1. By specifying the amount of memory when going TSR.

    Use this option if your system allows it.

2. By specifying the maximum number of connections the server will allow.

Use this option if you have 'heavy' web pages – usually the type where pages consist of frames and many images, etc. Connections are generally reset when more connections are attempted than the defined maximum. The client then must retry to establish the lost connections, leading to a more distributed load on the server.

Spawning CGI

An external program, indicated by the requested URL, is spawned. All relevant information is passed as environment variables. The program gets all input (e.g. posted data) from *standard in* and sends all response through *standard out.*

This type of CGI is discouraged in favor of *Extension CGI (3.1).*

Remarks on spawning CGI operation.

The following CGI environment variables are supported:

CONTENT_TYPE, CONTENT_LENGTH, PATH, COMSPEC and REQUEST_METHOD.

Enough free memory must be available when spawning a CGI program, or no swapping or overlaying will be attempted. Since COMMAND.COM uses all free memory, it follows that no CGI program will be spawned if COMMAND.COM is the current foreground program.

CGI programs must be small and must execute reasonably quickly. While a CGI program is executing, the HTTP server is effectively blocked and cannot service any other requests. No console input or output should be used. A CGI program is invoked by a URL containing a path of /cgi-bin/<cgi-program> where <cgi-program> is the name of an executable program which must be in the HTTP root directory or in the path. Note that the "/cgi-bin/" part is stripped off and does not represent a real directory. <Cgi-program> may be followed by a "?" and a command line. On entry to the CGI program, the environment variables listed above are set up and can be accessed. If a command line is given, it can also be accessed in the normal way.

The CGI program generates a dynamic page by writing to STDOUT. When the CGI program terminates, this output is sent to the remote client (browser). The output can consist of a header and a body part separated by an empty line. If the header contains a “Content-type:” line, the content type will be set to that type and only the body will be sent to the client. Otherwise all the output will be sent to the client using content type “text/plain”. COMMAND.COM can be invoked as a CGI program to perform simple DOS functions e.g. directory listings. The following example performs a directory listing:

http://www.embedded-server.com/cgi-bin/command?/cdir

The next one performs a wide directory listing using a wild-card specification:

http://www.embedded-server.com/cgi-bin/command?/cdir%20*.htm%20/w

Note the use of %20 to specify a space character.

Refer to the INDEX.HTM web page for an example of various ways of calling CGI programs. The NUM.EXE program with source code NUM.C, demonstrates the use of a header and body part building a simple “page visited” web page:

```
printf("Content-type: text/html\n\n”

”<html>\n<h1>\nThis page has been visited %d times\n</h1>\n",number);

        printf("<P><P><A HREF=\"/index.htm\">Back</A>.</html>\n");
```

Forms programming can be performed using either the GET or POST methods. When GET is used, form data is copied to the command line and is limited to 128 characters including the URL part. When the POST method is used, the command line is also built. In addition, form data are available from STDIN and is limited by disk space only. See the forms programming example consisting of FORM.HTM, FORM.EXE and FORM.C for examples of using both the GET and POST methods.

So that you may fully understand CGI programming, this detailed explanation of the server operation is provided.

Whenever HTTPD receives a URL containing “/cgi-bin/”, it interprets the rest of the URL as a DOS program to spawn and run to completion. The full path parsed from the URL is used, implying that the

program should be in physical directory called “/cgi-bin/” or a subdirectory thereof. E.g. “program.exe” should be in “%HTTP_DIR%\cgi-bin\” if the request is “GET /cgi-bin/program.exe”.

While this “CGI program” is executing, the server can accept new server connections, but will not respond to them before the CGI program terminates. The CGI program can be any DOS program that is small enough to fit into available memory. Since HTTPD is blocked while the CGI program executes, user interaction should not be used and the CGI program should complete in a reasonable time.

Operation on receiving a CGI URL:

If the CGI program name is followed by a "?", the rest of the line is sent as a command line to the CGI program after converting all %n combinations.

If a “Content-Type” header is encountered, the CONTENT_TYPE environment variable is set to the given value and if a “Content-Length” header is encountered, the CONTENT_LENGTH environment variable is set to the given value. The PATH and COMSPEC environment variables are copied to the new environment and the REQUEST_METHOD environment variable is set to either GET or POST.

If the POST method is used, the rest of the HTTP message is copied to a temporary file that is then re-directed to stdin. The stdout stream is redirected to another temporary file. After completion of the request, the temporary files are deleted. They will be created in the %HTTPTMP% directory.

The CGI program is now invoked. This program can check the environment variables, access the command line and in the case of a POST, read from stdin. All output that should be passed back to the HTTP client (Browser) is written to stdout. A single header line followed by an empty line, containing “Content-type: *content_type”* may be pre-pended to the data. This line will be used to set the content-type of the data being sent back. If such a header is not found, the content type will be set to “text/plain”.

Authentication

Default authentication matches the capabilities of the FTP server as documented in the SOCKETS SDK. A file called "SOCKET.UPW" should exist in the SOCKETS (environment variable) directory.

The default permission file controls remote console access. Each listed user has a single-letter privilege code set if he has privilege to use the Remote Console. The code should be missing if that user does not have Remote Console privilege.

An additional feature is implemented - **htaccess**.

This feature provides a per-directory permission override mechanism.

It is enabled using '/t' as command line switch.

If htaccess is enabled, the default mechanism may be skipped (but no default users or remote console access will be available).

A file called HTACCESS (typically hidden) contains authentication overrides to enable partial anonymous access or additional password security to subdirectories, etc. If this feature is activated, the server code will look for HTACCESS files in each directory starting from the requested path and

continuing upward in the directory structure (assuming the root directory to be at the top) until an HTACCESS file is found. If no file is found, then the default settings are used. An anonymous access entry is available for the developer to specify that some subdirectory is authorized for any user, although its parent directory is password-protected.

CGI scripts are also controlled via the HTACCESS mechanism.

Registers When Calling HTTPD Directly (not using the CGI adapter)

int HttpRegister(

far char **szPath*,

int (far *pfCallback)(HTTP_PARAMS far **psHttpParams*),

int *iFlags*,

DWORD *dwUserID*

);

Assembler level:

AH = 0

szName: DS:SI

pfCallback: ES:DI

iFlags: BX

dwUserID: CX:DX

Return in AX register.

Remarks about the callback:

The callback is guaranteed to only be called when DOS can be called. The DOS critical handler will be disabled and all critical errors will result in an access error without any user intervention. Since the callback happens at interrupt time, it should execute for as short a time as possible. After a done or error return, no further callbacks will be generated for the current request.

Only one callback will be active at any time. Calling an API function while executing the callback function will not result in another callback before the current callback has returned.

Note that the stack and the data segment on entry will be that of HTTPD. Depending on the memory model used for the extension and the amount of stack space required, it may be required to switch stacks during the callback.

int HTTPDeRegister(char **pszPath*);

AH = 1

pszPath: DS:SI

Return in AX register.

Remarks:

After this call no more callbacks will be generated for this URL. Any requests in progress will be terminated with an error to the peer. This API must be called for all registrations made by a program before terminating that program; otherwise the system will inevitably crash on any subsequent request.

int HTTPGETData(int *iHandle*, char far **pcBuf*, int *iCount*);

AH = 2

iHandle: BX

pcBuf: DS:SI

iCount: CX

Return in AX register.

int HttpSendData(int *iHandle*, char far **pcBuf*, int *iCount*);

AH = 3

iHandle: BX

pcBuf: DS:SI

iCount: CX

Return in AX register.

int HttpSubmitFile(int *iHandle*, char far **pszFileName*);

AH = 4

iHandle: BX

pszFileName: DS:SI

Return in AX register.

Command Line Switches when loading the Server

Any combination of these switches may be used. They should be separated by at least one space.

/? /h This help screen

/r - tsr

/s - status

/t - htaccess - enable directory level authentication

/u - unload if resident

/c - close listen

/g - allow old type (spawning) CGI

/p - Passive mode

/i=<int_num> Interrupt number for cgi API

/m=<bytes> - set memory size

/n=<maximum number of simultaneous connections>

/a=<columns>, <rows> - set screen aspect

/v=[: <offset>] - set video buffer address (hex)

The “root” directory for web content is the current directory when HTTPD is started. This can be changed by setting an environment variable HTTP_DIR e.g.

SET HTTP_DIR=D:\SERVER\WEB

Format of "SOCKET.UPW"

This is the same file used for the FTP server’s (*FTPD.EXE*) permissions.

This file consists of lines where each line contains a user's information.

A line starting with a # is considered a comment and is ignored.

Each line consists of four fields:

username

The name of this user. If it is *, it will be used when the client does not specify a username.

password

This user's password. If it is *, no password is required.

working directory

The user will only have access to this directory and its subdirectories

If it is ‘/’, this user has access to the whole system.

HTTP_DIR can be referred to as '\'.

If a relative path is specified, it is appended to HTTP_DIR.

permissions

IMPORTANT when a user is granted both FTP and HTTP permissions, the FTP permissions should appear **first**, otherwise they will be ignored.

Operations allowed. May contain any combination of the following tokens:

e - User may 'g**e**t' files

p - User may '**p**ost' files

g - User may use c**g**i

m - User may use Re**m**ote Console

Fields should be separated by single spaces.

If any field is missing the entry is ignored.

A comment may follow the last field (permissions) of the line.

Note: If a default user is supplied, it should always appear first in the list of users. Only users below the default user will be considered.

Format of "htaccess"

Any directory may contain this file, and serve as overrides to the general permissions for the containing directory and all its subs until another htaccess is found.

This file consists of lines where each line contains a user's information.

A line starting with a # is considered a comment and is ignored.

Each line consists of three fields:

username

The name of this user. If it is *, it will be used when the client didn't

specify a username.

password

This user's password. If it is *, no password is required.

permissions

Operations allowed. may contain any combination of following tokens:

e - User may 'get' files

p - User may 'post' files

g - User may use cgi

Fields should be separated by single spaces.

If any field is missing the entry is ignored.

A comment may follow the last field (permissions) of the line.

Note: If a default user is supplied, it should always appear first in the list of users. Only users below the default user will be considered.

SSI Definitions

```
/* SSI.H
 * SSI API functions
 */

/*
* initialize SSI adapter: hook HTTP server
 * pszCgiName - The name on which to react, e.g. cgi-bin/getpage.exe
 */
int CgiInit(char *pszCgiName);

/*
 * Unhook HTTP server. Function MUST be called before exiting, otherwise
 * system will crash.
 */
void CgiQuit(void);
/*
 * Add a variable and its value to the specified parameter set.
 */
```

```
int AddVar(int iHandle, char *pszName, char *pszValue);
/*
 * Set the filename of the specified handle
 */
int SetFile(int iHandle, char *pszFilename);

/*
* the only function the user of the SSI interface must implement
 *
 * This function will be called each time a new request arrives.
 * pszQuery will hold the query.
 *
 * A user must use the 'SetFile' and 'AddVar' functions to set the
* template file and add SSI variables. The provided iHandle should
 * be used when calling these functions
 */
void SetVariables(char *pszQuery, int iHandle);
```

Using the CGI Adapter (*CGIADAP.C*)

CGIADAP is an interface a user may utilize to implement external cgi programs. This interface performs stack and context switches, and provides ordinary C functions to access the http server (*HTTPD.exe*).

The interface is described by referring to its header file:

```
/* CGIADAP.H
 * contains definitions for cgi adapter interface
*/

//contains all the definitions for return values and error messages
#include "htapic.h"

/*
adapter for cgi interfacing with HTTPD server
*/
/*
int HttpRegister(char far *szPath,
 int (far *pfCallback)(HTTP_PARAMS far *psHttpParams),
 int iFlags, DWORD dwUserID);
```

Arguments:

szPath:

The string identifying a URL. It should be an exact match of the abs_path part of the URI minus the leading '/'. For instance, If you want to capture all http://myserver.com/cgi-bin/getpage.exe, you should register 'cgi-bin/getpage.exe'.

pfCallback:

The callback is guaranteed to only be called when DOS can be called.

The DOS critical handler will be disabled and all critical errors will result in an access error without any user intervention.

Since the callback happens at interrupt time, it should execute for as short a time as possible. After a done or error return, no further callbacks will be generated for the current request.

psHttpParams:

psHttpParams->iReason - reason for callback:

R_NEWREQ - New HTTP request. pszCommandLine points to the command line passed in the URL. The number contained in iValue specifies the HTTP operation; RQ_GET for GET and RQ_POST for POST.

R_INDATA - Input data available, iValue contains count.

R_OUTDATA - Can send output data, iValue contains count.

R_ENDDATA - Peer closed connection i.e. "end of input data"

R_CLOSED - Connection closed.

psHttpParams->iHandle

HTTPD handle, used in subsequent API calls for this request.

The user should not modify it.

See htapic.h for the other definitions

return values:

RET_OK - not done, give me more upcalls

RET_DONE - done, no more upcalls please

RET_ERR - done, error

Only one callback will be active at any time.

Calling an API function while executing the callback function will not result in another callback before the current callback has returned.

iFlags:

F_OVERLAP - overlapped request (1), non-overlapped request (0).

All other bits are reserved.

dwUserID:

value passed to HttpRegister(); this value is for use by the

extension, HTTPD does not modify it.

return: OK (0) - ok

< 0: One of the error messages (see htapic.h)

*/

int HttpRegister(char far *szPath,

int (far *pfCallback)(HTTP_PARAMS far *psHttpParams),

int iFlags, unsigned long dwUserID);

/*

int HTTPDeRegister(char *pszPath);

return: OK (0) - ok

< 0: - One of the error messages (see htapic.h)

Remarks:

After this call no more callbacks will be generated for this URL.

Any requests in progress will be terminated with an error to the peer.

This API must be called for all registrations made by a program before terminating that program; otherwise the system will inevitably crash on any subsequent request.

*/

int HTTPDeRegister(char far *pszPath);

/*

int HTTPGETData(int iHandle, char far *pcBuf, int iCount);

Remarks:

This API can be called when a POST operation has been indicated by the callback. If more data is expected and the extension is busy executing the callback function, a 0 return should be made from the callback indicating it is still busy and getting more data should be attempted at the next callback.

return: >= 0 - ok, bytes received

< 0: One of the error messages (see htapic.h)

*/

int HTTPGETData(int iHandle, char far *pcBuf, int iCount);

/*

int HttpSendData(int iHandle, char far *pcBuf, int iCount);

Remarks:

This API can be called to send data to the peer.

If the return indicates that less than the requested number of bytes has been sent and the extension is busy executing the callback function, a 0 return should be made from the callback indicating it is still busy. Then an attempt to send more data should be made at the next callback.

All the required data should be sent to the peer before the HttpSubmitFile() function is used. After HttpSubmitFile(), HttpSendData() should not be called again.

return: >= 0 number of bytes actually sent

< 0 one of the error messages (see htapic.h)

*/

int HttpSendData(int iHandle, char far *pcBuf, int iCount);

/*

int HttpSubmitFile(int iHandle, char far *pszFileName);

Remarks:

The extension can submit a file to be sent to the peer in response to a request. The file will be logically appended to any data already sent using HttpSendData(). The file should not be exclusively opened when it is submitted. After it is transmitted, transmit upcalls will be issued normally. This gives the user the ability to send any number of files on the connection with arbitrary data in between.

return: 0 OK

< 0 error

*/

int HttpSubmitFile(int iHandle, char far *pszFileName);

/*

* TAKE NOTE - STACK POINTER:

*

* The stack pointer for callbacks is by default set to _SP - 1000, the first

```
 * time the HTTP API is called. If you
 * would need space on the stack, or for some reason want to make it tighter,
 * set the stack pointer for callbacks manually. Be careful not to write over
 * used memory.
*/
int GetStackPointer(void);
void SetStackPointer(int iPointer);
```

Constants and Definitions used by CGI API

```
/* HTAPIC.H
 * Header file for cgiadap.h
 * CONSTANT DEFINITIONS used by API
*/
//API functions (set in AH when API called)
#define APIF_REGISTER           0
#define APIF_DEREGISTER                 1
#define APIF_GETDATA            2
#define APIF_SENDDATA                   3
#define APIF_SENDFILE           4
#define APIF_GETVERSION                 5
//flags
#define F_OVERLAP               1//indicate multiple requests at a time


/* Error return code */
/* NOTE in all API calls, the negative of these functions is returned */
#define  NONE                   0       /* No error */
#define  NO_ERR                 0       /* No error */
#define  OK                     0       /* No error */
#define  CON_EXISTS             1       /* Connection already exists */
```

```
#define  NO_CONN            2       /* Connection does not exist */
#define  CON_CLOS           3       /* Connection closing */
#define  NO_SPACE           4       /* No memory */
#define  WOULDBLK           5       /* Would block */
#define  NOPROTO            6       /* Protocol or mode not supported */
#define  INVALID            7       /* Invalid arguments */
#define  BUFSHORT           8       /* Buffer too short for data */
#define BADFILENAME         9       /* cant open file */
#define NO_USER                     11      /* User doesn't exist */
#define ERR_DUPL                    10      /* duplicate name (name already exists) */
#define  RE_ENTER           27      /* Re-entry of SOCKETS */
#define NO_API              50      /* API not loaded or invalid API call */
#define ERR_EOF                     51      /* end of stream */


//http request types
#define RQ_NONE             0               // unknown
#define RQ_GET              1               // get
#define RQ_POST             2               // post
#define RQ_CGI              3               // cgi
#define RQ_REMCON           4               // remote console


//callback:
//parameters
typedef struct HTTP_PARAMS_S
{
        int iReason;
        int iHandle;            //handle for request
```

```
        long dwUserID;          //user defined ID (set at register)
        long dwSessionID;       //per connection ID, set at any upcall by user
        int iValue;
        long lContentLen;
        char far *szQuery;
} HTTP_PARAMS;

//callback function
//int (far *pfCallback)(HTTP_PARAMS far *psHttpParams);

//return values:
#define RET_OK          0       //give me more upcalls
#define RET_DONE        1       //done, don't want any more data
#define RET_ERR         -1      //I experienced an error

//reasons (iReason one of)
#define R_NEWREQ            0       //new request
#define R_INDATA            1       //may read data
#define R_OUTDATA           2       //may transmit data
#define R_ENDDATA           3       //end of stream reached
#define R_CLOSED            4       //connection closed
```

Chapter 11, managing the Network and Troubleshooting

This chapter describes solutions to common LAN problems, both configuration and performance.

Network Management

The Network Manager is expected to:

- Setup each SOCKETS application according to user requirements.
- Monitor the status of various connections and trace traffic as it traverses the network, in order to resolve problems.
- Modify the software configuration to align it with changes in the physical environment on which it runs.

Configuration Case Studies

Managing Host Names on a File Server-Based LAN

The SETHOST.EXE program manages the host names on a file server based LAN. The purpose of running SETHOST.EXE is to keep all the IP addresses in a single file on the server and allow all workstations to run the same software setup and yet maintain a unique IP address at each workstation. An alternative for non-file server-based networks is to use BOOTP or DHCP where a suitable server is available.

Installing SETHOST

Edit the HOSTS file to add all the workstation names and IP addresses. We recommend that all the workstation IP addresses be preceded with an asterisk to make them hidden to other users looking into the list of hosts. For example,

```
*198.147.35.120    admin03
```

Copy the SETHOST.EXE program and an empty file MACHOST.MF to your server in a directory that is named, for this example, X:\SOCKETS.

At each workstation, log in as supervisor (or have write access to X:\SOCKETS) and execute the DOS commands:

```
x:
cd \SOCKETS\DOS
SETHOST /N=WS_NAME
SETHOST
SET
```

The SETHOST /N=WS_NAME command creates an entry in the MACHOST.MF file with the name of the workstation and its MAC (Ethernet board) address. This is the important “once-only”

command. The next two commands are to verify that the *ws_name* is stored in the environment variable HOSTNAME.

In the AUTOEXEC.BAT, or any batch file executed after login and before running SOCKETS, put the following commands:

```
x:
CD \SOCKETS\DOS
SETHOST
```

In SOCKETS, set the IP address in SOCKET.CFG as follows:

```
ip address \HOSTNAME\
```

The IP addresses are now linked to the MAC addresses of the network cards. If you change a network board or swap a PC, must update the MACHOST.MF file with

```
sethost /ws_name
```

The MACHOST.MF file keeps the mapping from the MAC addresses to the workstation names and the HOSTS is used to map the workstation name to its IP address.

The variable name HOSTNAME and filename MACHOST.MF are the defaults for SETHOST.EXE but they can be user specified. Execute sethost /? to see the available options. See also “SETHOST, IP Address Maintenance Utility

.

Advanced Network Configuration

SOCKETS offers additional networking capabilities for large networks. Using the Routing Information Protocol (RIP), SOCKETS can be configured to be aware of multiple IP routers/gateways.

BOOTP servers are detected when SOCKETS is started without specifying an IP address or an address of 0.0.0.0. To trace the BOOTP negotiations, use a trace all **iodt** command in your .CFG file before defining any interfaces.

DHCP servers are detected and used when SOCKETS is started with an IP address of 0.0.0.1.

For file server linked networks the SOCKETS utility SETHOST can be used to centralize management of IP addresses. SETHOST maintains a file mapping of the Ethernet (MAC) address of each machine to an IP address.

SOCKETS’ Alternative Routing feature allows more than one route to be specified to a particular host or network. Failure of one route causes an automatic switch to the next route. The failed route is tested periodically and used again when it becomes available.

Tuning TCP/IP

Tuning a computer is a trade-off between the speed of operation and the amount of memory it uses. Performance depends on the number of TCP connections for the computer; more sessions require more memory.

TCP Retry Strategy

If there is a delay in your network connections, SOCKETS employs an intelligent retry strategy. A retry is attempted after a retry interval that gets longer as a function of the number of the retry. The first retry is attempted after the current RTT (Round Trip Time) plus one PC clock tick (18 milliseconds) has elapsed without a response. SOCKETS calculates the RTT as a smooth average of past measured RTTs, starting with the IRTT on a new connection.

For the first five intervals, the time doubles, giving interval lengths of 2, 4, 8, 16 and 25 times the RTT. Then, the square of the interval number is used starting with 5-squared, giving 25 times RTT. Consequently, increasing the number of retries can cause the total elapsed time to become quite long. More than 255 retries results in an infinite number of retries, causing connections to never time out.

When a SOCKETS station starts up, it sends a broadcast ARP request with its IP address to check for duplicate IP addresses. Any SOCKETS client in a retry mode picks up this ARP. Then it retries immediately without waiting for the next scheduled retry time.

Number of retries	1	2	3	4	5	6	7	8	9	10	11	12
Interval length (in RTT)	1	2	4	8	16	25	36	49	64	81	100	121
Total time (in RTT)	1	3	7	15	31	56	92	141	205	286	386	507

To get the current RTT in use for a connection n, use the **tcp status n** command that gives the smoothed average RTT indicated by SRTT.

Keep-alive

All SOCKETS servers test established connections after a minute of no-traffic by sending an empty packet with a decreased sequence number and waiting for the acknowledgement. If the server does not receive an acknowledgement it retries using the retry strategy described above with the smoothed Round Trip Time (RTT). When all the retries have failed, the connection has timed out and the server resets that connection. This prevents servers from hanging in a listen state when the remote client has stopped responding.

Troubleshooting

Problems with LICENSE.DAT File

For SOCKETS demo only (socketmd.exe and socketpd.exe).

The SOCKET environment variable is required in the demonstration version to indicate the directory with the LICENSE.DAT file, which contains the license information.

The LICENSE.DAT file contains your license information in demonstration versions. When this error message appears, the most frequent problem is that the file is in the wrong directory. SOCKETS looks in the SOCKETS directory (as given in the SOCKET environment variable) for the LICENSE.DAT file and then (to accommodate simple new installations) in the \SOCKETS directory of the current drive. If the file contents have been damaged, it will not run. The information in the LICENSE.DAT file is displayed when you start. The number next to SL indicates the number of concurrent users allowed.

XPING

The XPING utility is the most basic test to see if connections are working. Always first try to ping a host that does not seem to respond. Failure to get a response to a ping can usually be traced to one of the following reasons:

- Network cable not plugged in.
- Incomplete bind of the drivers on the local machine. (Carefully check all of the diagnostic messages while booting).
- Inadequate routing information. It may also mean an invalid return route somewhere.
- Lost packages along the route. Sending many ping requests and get only some back; could relate to network hardware problems.
- Remote host is not responding (not switched on, software not loaded, not connected, and so on).

A ping response displays the time taken to get a response. Note that the timing depends on the clock ticks. In an 80x86 PC these ticks are approximately 55ms, so limit the accuracy of the reported times to 55ms.

Utility Programs

The utilities described in the following sections will help you to configure and test your SOCKETS installation.

PRNTEST, Printer/Port Test Utility

The PRNTEST utility, located in the DOS\TESTS directory, checks the response from the selected parallel port where LPT1 = 0; LPT2 = 1 and LPT3 = 2. PRNTEST shows the returned BIOS code from the printer port. The meaning of the status bits from the BIOS may differ between printers and BIOSes. The output displays the status of all the bits with their respective (normal) meanings. The bits are anded to give the status mask that can be used with the socket print server under DOS. The status is refreshed to reflect any changes caused by creating different error conditions.

PRNTEST

Syntax

prntest [0 | 1 | 2 | 3]

Example

```
prntest 0
```

Remarks

A typical printer-ready result could appear as follows:

Printer status = 0x90

Bits positions:

0x80: [1] Ready

0x40: [0] Not Acknowledged

0x20: [0] Paper supply OK

0x10: [1] Selected

0x08: [0] No I/O Error

0x04: [0] Not used

0x02: [0] Last character has not been sent to printer

0x01: [0] No Timeout

Testing port LPT1:

The value in the square brackets [ ] indicates the state of the relevant bit. Error conditions are flagged in red, although the red color does not always correspond to the bit status, but varies from bit to bit. The Printer status value at the top indicates the total of the set bits.

Normally for a printer to print, three conditions must be met: The printer must

Be selected

Be ready

Have paper.

Sum the hexadecimal values for these conditions and use that as the **/s** and **/m** parameters for the **start prntserv** command. The default values used are /m=A8 for the mask and /s=80 for the status, which imply that data is sent to the printer when it is not busy and 'Out of paper' or '/O Error' status are not set. 'Printer selected' is ignored, as this status bit is often not reported correctly. To include the 'Printer selected' bit, specify /m=B8 and /s=90.

PDTEST, Packet Driver Test Utility

PDTEST

The PDTEST utility is a diagnostic program that tests loaded packet drivers. PDTEST does not perform a full test since it does not transmit traffic through the drivers, but just checks their status. It reports such things as the interrupt vector of the packet driver, its class, the MAC address, and

information on the status of the driver. PDTEST is useful for checking that the packet driver was actually loaded which interrupt was used, and the class of packet driver. Classes supported by SOCKETS are marked with an asterisk (*) in the following list of recognized classes:

Class 1*	DIX Ethernet_II
Class 3*	802.5 Token Ring
Class 5	Appletalk
Class 6*	SLIP
Class 9	AX.25 Amateur Radio
Class 11*	802.3 with 802.2 headers IEEE
Class 12	FDDI with 802.2 headers
Class 13	Internet X.25
Class 14	Northern Telecom LANSTAR encapsulating DIX
Class 16	Point to Point Protocol for serial lines
Class 17	802.5 Token Ring w/expanded RIFs

Most Ethernet packet drivers support both classes 1 and 11. Class 1 is the default class and should normally be used.

Syntax

pdtest

Example (output)

```
Packet driver found at 0x60
Version 9, class 1, type 57, number 0, functionality 6
Name: MAC/DIS converter
High performance driver
Rev 1.09 par_len 14 add_len 6 mtu 1514 multicast_buf 0
Rcv_bufs 0 xmt_bufs 0 int_num 0x0
Address: 00:00:c0:08:d7:15
Extended driver
Packets:in 511 out 595 bytes:in 112664 out 74476
Errors:in 0 out 0 packets lost 0
```

SETHOST, IP Address Maintenance Utility

SETHOST

The SETHOST utility sets an environment variable (default HOSTNAME) to the name contained in a map file (default MACHOST.MF) according to the hardware address (MAC or Ethernet) found by

searching for a packet driver tsr. Network management is simplified when using %HOSTNAME% in the SOCKET.CFG files to set IP addresses.

SETHOST can be used in either of two modes:

- To update the MACHOST.MF file or
- To set the HOSTNAME environment variable.

Syntax

sethost [/f=*n* | /n=*hostid*] [/c=] [/m=*map_file*] [/v=*variable*]

Options

without a *hostid* set the variable

/f=*n*
: Use *n*th environment block (required for some systems - try /n=1 first.)

/n=*hostid*
: This is the IP address of the local PC in symbolic form as in HOSTS or in decimal form to add to the mapfile

/c=
: Preserve the case of the environment variable

/m=*map_file*
: Use map_file instead of default MACHOST.MF

/v=*variable*
: Use variable instead of the default name: SETHOST

Note: The equal signs are required in this case since SETHOST supports applications where the n= is optional. A slash without an equal sign indicates the setting of a host id.

Example

To modify or add an entry in MACHOST.MF:

```
sethost /n=ws_name
```

To set the variable HOSTNAME:

```
sethost
```

Installing SETHOST

Installing SETHOST requires a working installation of SOCKETS. Also, all workstations run SOCKETS from a server disk.

Edit the HOSTS file to add all the workstations' names and IP addresses. We recommend that all the workstation IP addresses be preceded with an asterisk to make them hidden to other users looking into the list of hosts. For example, *198.147.35.120 admin03

Copy the SETHOST.EXE program and an empty file MACHOST.MF to your server, in a directory such as X:\SOCKETS, for example.

At each workstation, log in as supervisor (or have write access to X:\SOCKETS) and execute the DOS commands:

```
x:
```

```
CD \SOCKETS
SETHOST /N=WS_NAME
SETHOST
SET
```

The SETHOST /N=WS_NAME command creates an entry in the MACHOST.MF file with the name of the workstation and its MAC (Ethernet card) address. This is the important "once-only" command. The next two commands are to verify that the ws_name is stored in the environment variable HOSTNAME.

In the AUTOEXEC.BAT or any batch file executed after login and before running SOCKETS, put the following commands:

```
x:
CD \SOCKETS
SETHOST
```

The IP addresses are now linked to the MAC addresses of the network cards. If a network adapter or host system is changed, update the MACHOST.MF file with

```
sethost /ws_name
```

The MACHOST.MF file keeps the mapping from the MAC addresses to the workstation names and the HOSTS file maps the workstation name to its IP address.

The variable name HOSTNAME and filename MACHOST.MF are the defaults for SETHOST.EXE but can be user-specified. Run SETHOST /? to display the available options.

IPSTAT, IP and Memory Statistics Utility

IPSTAT

The IPSTAT utility returns statistics on IP and memory. Use IPSTAT to check for error conditions and memory problems.

Syntax

ipstat

Example Output

IP stats at 160F:04C8:

Total packets	2671
Smaller than minimum size	0
IP header length too small	0
Wrong IP version	0
IP header checksum errors	0
Unsupported protocol	0
Memory available	9016
Memory allocation failures	0

Memory free errors	0
Minimum stack observed	886

Glossary/Definitions

ACK (Acknowledgment flag; TCP header)

A response sent by a receiver to indicate successful reception of information. Acknowledgements may be implemented at any level including the physical level (using voltage on one or more wires to coordinate transfer), at the link level (to indicate successful transmission across a single hardware link), or at higher level (e.g., to allow an application program at the final destination to respond to an application program at the source).

Address Mask (also referred to as NetMask)

A bit mask used to select bits from an IP address for subnet addressing. The mask is 32 bits long, and selects the network portion of the IP address and one or more bits of the local portion.

ANSI (American National Standards Institute)

A group that defines U.S. standards for the information processing industry. ANSI participates in defining network protocol standards.

API (Application Program Interface)

An API is a specification of the methods an application programmer can use to access services provided by a software module. In the case of a network, the API specifies the interface to the network software.
In TCP/IP, the idea of a "Socket" as the endpoint of a connection is used. A "socket" then refers to an abstraction to define the endpoint of a connection as far as the API is concerned. A socket can be created, opened, read, written, closed, and deleted in much the same way a file is handled in DOS. The difference is that two sockets must exist, normally on two hosts, before a connection can be made. A read operation on one side must always have a matching write operation on the other side.
A common way of interfacing a terminal emulator to networking software in a PC is to use Interrupt 14h. This is the PC BIOS entry point for serial port support, but when used for networking purposes, the original entry point is reused to provide a similar, but much expanded function. In addition to the native character at a time transfer, block transfers are also offered to increase throughput.

ARP (Address Resolution Protocol)

The TCP/IP protocol used to dynamically bind a high-level IP Address to a low-level physical hardware address. ARP is used across a single physical network and is limited to networks that support hardware broadcast.

Asynchronous MUX

A MUX server is a network computer with one or more asynchronous ports connected to serial devices that normally have terminals connected to them, for example, asynchronous host ports, serial printers or modems. Any workstation in the network can be used to log into the serial device using Telnet.

Baud

Literally, the number of times per second the signal can change on a transmission line. Commonly, the transmission line uses only two signal states making the baud rate equal to the number of bits per second that can be transferred. The underlying transmission technique may use some of the bandwidth, so it may

not be the case that users experience data transfers at the line's specified bit rate.

BOOTP (Bootstrap Protocol)

A protocol a host uses to obtain startup information, including its IP address, from a server.

Broadcast

A packet delivery system that delivers a copy of a given packet to all hosts that attach to it is said to broadcast the packet. Broadcast may be implemented with hardware or software.

CSLIP (Compressed Serial Line Internet Protocol)

CSLIP is an enhancement of SLIP by implementing Van Jacobson header compression. CSLIP uses more memory than SLIP but provides better throughput and faster response times, especially on small packets.

Datagram

The basic unit of information passed across a TCP/IP connection. An IP datagram is to an Internet as a hardware packet is to a physical network. It contains a source and destination address along with data.

DHCP (Dynamic Host Configuration Protocol)

A protocol that a host uses to obtain all necessary configuration information including IP address.

DNS (Domain Name Server)

The on-line distributed database system used to map human-readable machine names into IP addresses. DNS servers throughout the connected Internet implement a hierarchical namespace that allows sites freedom in assigning machine names and addresses. DNS also supports separate mappings between main destinations and IP addresses.

Domain

A part of the DNS naming hierarchy. Syntactically, a domain name consists of a sequence of names separated by periods.

Flow control

Control of the rate at which hosts or routers inject packets into a network or Internet, usually to avoid congestion.

FTP (File Transfer Protocol)

The TCP/IP standard, high-level protocol for transferring files from one machine to another. FTP uses TCP.

Gateway

Originally, researchers used the term IP gateway for dedicated computers that route packets; vendors have adopted the term IP router. Gateway now refers to an application program that interconnects two services.

ICMP (Internet Control Message Protocol)

An integral part of the Internet Protocol that handles error and control messages. Specifically, router and hosts use ICMP to send reports of problems about datagrams back to the original source that sent the

datagram. ICMP also includes an echo request/reply used to test whether a destination is reachable and responding.

IPCP (IP Control Protocol)

A PPP protocol responsible for configuring the IP protocol parameters on both ends of the point-to-point link.

IPCPIN

A PPP protocol monitoring incoming requests responsible for configuring the IP protocol parameters on both ends of the point-to-point link. This was implemented to allow one instance of SOCKETS to act as both a client and server.

IP (Internet Protocol)

The TCP/IP standard protocol that defines the IP datagram as the unit of information passed across an Internet and provides the basis for connectionless, best-effort packet delivery service. IP includes the ICMP control and error message protocol as an integral part. The entire protocol suite is often referred to as TCP/IP because TCP and IP are the two fundamental protocols.

LAN (Local Area Network)

Any physical network technology designed to span short distances (up to a few thousand meters). Usually, LANs operate at tens of megabits per second through several gigabits per second.

LCP (Link Control Protocol)

A PPP protocol responsible for establishing, configuring, and testing the data link connection.

LCPIN

A PPP protocol monitoring incoming requests which is responsible for establishing, configuring, and testing the data link connection.

MIME (Multipurpose Internet Mail Extensions)

A standard used to encode data such as images as printable ASCII text for transmission through e-mail.

Modem

A modem (modulator/demodulator) converts digital computer signals into analog signals as used in telephone equipment. The data is sent across the telephone lines and converted back to digital signals by another modem at the destination node.

Using dial-up modems, a remote client can gain access to a network through the telephone line. Remote client users can gain access to the network resources just as if they were physically connected to the LAN. SOCKETS supports the PPP, SLIP and CSLIP protocols.

MSS (maximum segment size)

The largest segment allowed for communicating across a TCP/IP connection.

MTU (maximum transmission unit)

The largest amount of data that can be transferred across a given physical network.

Multicast

A technique that allows copies of a single packet to be passes to a selected subset of all possible destinations.

Nagle Algorithm

This algorithm states that under some circumstances, there will be a waiting period of 200 ms before data is sent over a connection.. The following are the specific rules used by the Nagle Algorithm in deciding when to send data:

- If a packet is equal or larger than the segment size (or MTU), and the TCP window is not full, send an MTU size buffer immediately
- If the interface is idle, or the TCP_NODELAY flag is set, and the TCP window is not full, send the buffer immediately.
- If there is less than ½ of the TCP window in outstanding data, send the buffer immediately.
- If sending less than a segment size buffer, and if more than ½ the TCP window is outstanding, and TCP_NODELAY is not set, wait up to 200 msec for more data before sending the buffer.

For more information please see RFC-896, “Congestion Control in IP/TCP”

Packet

Used loosely to refer to any small block of data sent across a packet switching network.

Packet Driver

Local area network software that divides data into packets for sending on the network, and reassembles the data into its original form when it arrives at its destination.

PPP (Point-to-Point Protocol)

A protocol for framing IP when sending across a serial line.

RDP (Reliable Datagram Protocol)

A protocol that provides reliable datagram service on top of the standard unreliable datagram service that IP provides. RDP is not among the most widely implemented TCP/IP protocols.

RFC (Request for Comment)

The name of a series of notes that contain surveys, measurements, ideas, techniques, and observations, as well as proposed and accepted TCP/IP protocol standards.

RIP (Routing Information Protocol)

A protocol used to propagate routing information inside an autonomous system.

Router

A special purpose, dedicated computer that attaches to two or more networks and forwards packets from one to the other.

RPC (remote procedure call)

A technology in which a program invokes services across a network by making modified procedure calls.

RTT (round-trip time)

A measure of delay between two hosts.

SLIP (Serial Line Internet Protocol)

A framing protocol used to send IP across a serial line. SLIP is popular when sending IP over dialup phone lines.

SMTP (Simple Mail Transfer Protocol)

The TCP/IP standard protocol for transferring electronic mail messages from one machine to another.

TCP (Transmission Control Protocol)

The TCP/IP standard transport level protocol that provides the reliable, full duplex, stream service on which many application protocols depend.

TTL (time-to-live)

A technique used in best-effort delivery systems to avoid endlessly looping packets.

UDP (User Datagram Protocol)

The TCP/IP standard protocol that allows an application program on one machine to send a datagram to an application program on another.

WAN (wide area network)

Any physical network technology that spans large geographic distances.

WWW (World Wide Web)

The large-scale information service that allows a user to browse information. WWW offers a hypermedia system that can store information as text, graphics, audio, etc.

Index

S

T

U

W

X