The Transport Control Protocol - Internet Protocol (TCP-IP) family of networking protocols was developed under the auspices of the Department of Defense to standardize communication mechanisms within Department of Defense networks such as the ARPANET.

The protocols are documented in a collection of working papers known as Requests for Comments (RFCs). Appropriate RFC numbers appear throughout this document as new protocols are introduced.

Requirements

TCP-IP has both hardware and software requirements.

Hardware

- Ethernet
- Cooperating host (yours or theirs)
- 110X/118X with an Ethernet controller (usually co-resident on an otherwise inhabited module)
- XCVR interface cable
- XCVR installed on an Ethernet with a logical (direct or internet) connection to the cooperating host.

Software

You need the files enumerated in the section titled "Interlisp Files." Files loaded by the high-level modules TCPFTP, TCPFTPSRV, TCPCHAT, and TCPTFTP automatically load their dependencies. If you load files from floppy, you must load their dependencies first:

File	Dependencies
TCP	TCPLLIP
TCPCHAT	TCP, CHAT
TCPCONFIG	None
TCPDEBUG	TCP
TCPDOMAIN	TCPUDP
TCPFTP	TCPNAMES, TCP
TCPFTPSRV	TCPFTP
TCPHTE	None
TCPLLAR	None
TCPLLICMP	None
TCPLLIP	TCPHTE, TCPLLICMP, TCPLLAR

TCPNAMES	None
TCPTFTP	TCPUDP
TCPUDP	TCPLLIP

User Interface

TCP does not have a user interface module of its own. Its functions and variables are accessible via an Interlisp Executive, and you can direct some of its debugging information to a window.

As a network protocol module, it extends capability to other programs which may have their own window interfaces, for example, Chat and FileBrowser.

Installation

The first step in installing TCP-IP is to add your workstation to a network supporting TCP-IP and communications with others on the net. The rest of this section contains a step-by-step set of directions for this installation.

After you are on the network, load the required .LCOM modules for the type of service you want. For a full description of these modules, see the section "Interlisp Files."

Module	Implementation
TCPFTP	TCP-based file transfer protocol
TCPFTPSRV	TCP-based FTP server
TCPCHAT	TELNET protocol for the Chat system.
TCPTFTP	TFTP protocol.

Obtaining Network Addresses

The first thing you need to do is to get a TCP-IP address assigned to each of your workstations from your network administrator. If your site supports Domains, get the name of your local domain and the addresses of your domain server(s) from your network administrator. You will also need to know the network addresses and operating system of the hosts you want to communicate with and the addresses of any network gateways you have.

Note: The maximum length of the domain and organization fields is 20 characters each.

Be sure to find out whether your net is a true Class A, B or C network and is not broken up into subnets. If it is broken up into

subnets, be sure to read the discussion on SUBNETMASKs in "A Primer on IP Networks."

Warning For Sun Installations: When running TCP-IP to a Sun from an 11xx, directory enumeration on an unmatched directory path returns a listing for the top-level directory of the logged-in user. The TCPFTP protocol does not support directory creation.

Creating HOST.TXT File

Create a HOSTS.TXT file containing entries for the TCP-IP hosts needed by the user community and place a copy of the file on either a directory contained in the DIRECTORIES search path of each workstation on the net or the local disk of each Interlisp workstation.

The following is a sample HOSTS.TXT file:

```
; Hosts.txt,
; Internet Hosts Table for Networks 192.20.10.0 and 174.23.0.0
; 12-Dec-86
;
; The format of this file is documented in RFC 810, "DoD Internet
; Host Table Specification", which is available online at SRI-NIC
; as the file
;                [SRI-NIC]<RFC>RFC952.TXT
;
; It may be retrieved via FTP using username ANONYMOUS with
; any password.
;
; or as the file
;                                        [INDIGO]<RFC>RFC952.TXT
; Read access to GV World. Valid GV credentials required.
;
; The format for entries is:
;
; GATEWAY: ADDR, ADDR : NAME : CPUTYPE : OPSYS : PROTOCOLS :
; HOST: ADDR, ALTERNATE-ADDR (if any): HOSTNAME,NICKNAME : CPUTYPE :
;   OPSYS : PROTOCOLS :
;
; Where:
;;  ADDR = internet address in decimal, e.g., 26.0.0.73
;;  CPUTYPE = machine type (Xerox-11xx, VAX-11/780, SUN, etc.)
;;   OPSYS = operating system (UNIX, TOPS20, TENEX, VMS, Interlisp,
etc.)
;;  PROTOCOLS = transport/service (TCP/TELNET, TCP/FTP, etc.)
;;  : (colon) = field delimiter
;;  :: (2 colons, NO space between) = null field
;
```

```
HOST : 192.20.10.1 : Bach : Xerox-1108 : Interlisp : TCP/TELNET,
TCP/FTP :
HOST : 192.20.10.3 : PARC-VAXC : VAX-11/780 : UNIX : TCP/TELNET,
TCP/FTP :
HOST : 192.20.10.15 : Oberon : VAX-11/780 : VMS : TCP/TELNET, TCP/FTP
:
HOST : 192.20.10.71 : Explorer : TI-EXPLORER : TOPS-20 : TCP/TELNET,
TCP/FTP :
HOST : 174.23.77.22 : Sunrise : SUN : UNIX : TCP/TELNET, TCP/FTP :
HOST : 174.23.30.21 : Rutgers : VAX-11/780 : TOPS-20 : TCP/TELNET,
TCP/FTP :
HOST : 174.23.76.21 : Simba : SYMBOLICS : SYMBOLICS-3600 :
TCP/TELNET, TFP/FTP :
GATEWAY : 192.20.10.240, 174.23.77.250 : Hellsgate : VMS : IP/GW :
```

This example shows a host table that indicates that there are four hosts (Bach, PARC-VAXC, Oberon, and Explorer) on net 192.20.10.0, three hosts (Sunrise, Rutgers, and Simba) on net 174.23.0.0 and a gateway (Hellsgate) that connects the two.

In regard to the OPSYS field in the HOSTS.TXT file, it is preferable to use values recognized by the Lisp variable NETWORKOSTYPES. Interlisp is the default value if a host's OSType is not declared.

Note that if any host is accessible via another network protocol (for example, PUP or NS), you may desire to call the host by an unambiguous name when it is accessed via TCP. You can do this by giving it an unambiguous name in the HOSTS.TXT file.

If you ever modify the HOSTS.TXT table after TCP.LCOM has been loaded, use the function (\HTE.READ.FILE *'HOSTTABLE*) to reread the file.

For example,

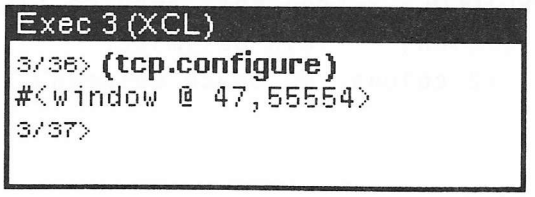
```
(\HTE.READ.FILE '{DSK}<LISPFILES>HOSTS.TXT)
```

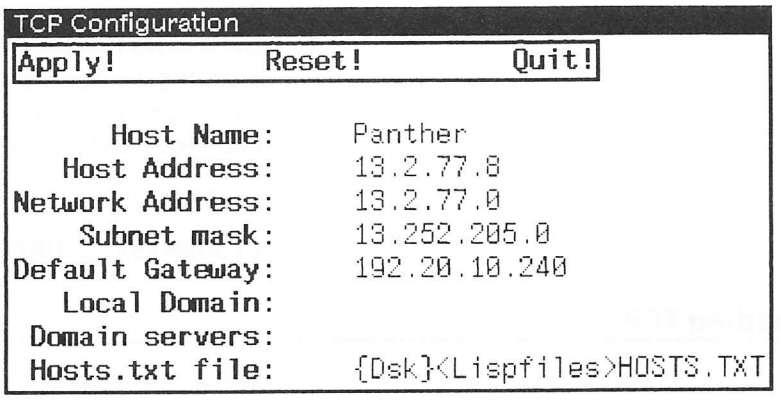
TCP.ALWAYS.READ.HOSTS.FILE [Variable]

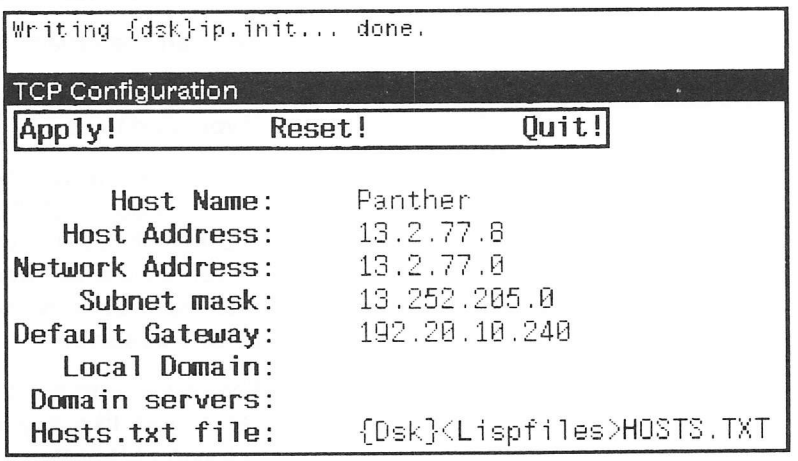
Initially set to T. Setting it to NIL causes the system to parse the HOSTS.TXT file only when the filename (stored in the configuration file) is different from the previously read filename, or the write date of the file has changed. The HOSTS.TXT file will always be read at least once when loading the software into a clean sysout.

Creating the Local IP.INIT File

TCP.CONFIGURE brings up a menu that you complete.







If any field does not apply to your site, leave it blank.

Selecting **Apply!** writes the file {DSK}<LISPFILES>IP.INIT to the local disk.

Note: The file {DSK}<LISPFILES>IP.INIT must exist on each Interlisp machine before TCP.LCOM is loaded. And this file *must* remain on the workstation and must not be copied to other workstations. Also, the font GACHA 12 MRR must be available.

Selecting **Reset!** resets the menu to the original state.

Selecting **Quit!** closes the window.

You must perform the TCP.CONFIGURE step individually on each workstation, but you need to perform it only once. As long as there is an IP.INIT file on the workstation, the TCP-IP module will be configured automatically whenever it is loaded or initialized.

If you change your IP.INIT file while TCP-IP is running, you will be prompted to confirm **Restarting TCP**. In most cases, you should confirm the restart.

Adding Host and Operating System Names to NETWORKOSTYPES

The variable NETWORKOSTYPES is used during Chat to determine the sequence of characters to send when performing auto-login. There should be an entry in NETWORKOSTYPES for each TCP host that you want to communicate with in the form (TCPHOSTNAME . OSTYPE).

For example:

```
((SUNRISE . UNIX)(RUTGERS . TOPS-20) etc)
```

Loading TCP

Make sure the variables DIRECTORIES and LISPUSERSDIRECTORIES point to the location of the .LCOM files of TCP-IP, or that they are in the connected directory.

You can then load TCP.LCOM which in turn loads its dependent files.

If you plan to do TCP file transfers, load TCPFTP.

If you plan to use the 11xx Lisp workstation as a TCPFTP Server host, load TCPFTPSRV. To start the server, evaluate (TCPFTP.SERVER). An Interlisp machine running the TCPFTP server should be identified as a TOPS-20 machine in the other Interlisp machines' HOSTS.TXT table. It will thus masquesrade as a TOPS-20 server.

The rest is automatic. You can treat an Interlisp host running the server just like any other TCPFTP server. The default path for resolving filenames is {DSK}<LISPFILES>, but you can change or override it.

For example, assume {ERIC} is a machine running the FTP server. From another machine which has TCPFTP loaded, you can do SEE {ERIC}{FLOPPY}FOO, which will type out the file FOO located on the floppy drive of {ERIC}.

If you plan to Chat to a TCP host, load CHAT, CHATTERMINAL, DMCHAT and then TCPCHAT.LCOM. Be sure that hosts with which you wish to chat have their NETWORKOSTYPES set.

If you plan to use the TCP Trivial File Transfer Protocol, load TCPTFTP.

Interlisp's TCPTFTP also provides a TCPTFTP server. Load TCPTFTP.LCOM and evaluate (TFTP.SERVER) . You can then use the appropriate TFTP commands to copy files from the Interlisp machine; for example TFTP.PUT and TFTP.GET.

Verifying TCP Connections

Load TCPDEBUG. Execute (TCPTRACE T) and you will be prompted to open a window to show TCP packets. Select INCOMING, OUTGOING and CONTENTS from the window's menu. If the host that you are communicating with has a TCP echoserver process you can then try (TCP.ECHOTEST 'HOSTNAME

3) . For example, using the above HOSTS.TXT file this would be (TCP.ECHOTEST 'SIMBA 3).

You will be prompted to open a window for the echo test and should see text, for example:

This is byte number 21
This is byte number 45
This is byte number 69

You should also see packets being sent and received in the TCPTRACE window.

Note: If a remote host is not running a TCP echo server process you will not get this response.

Connecting, Transferring Files, and Chatting to a Host

First log in to the host by typing (LOGIN 'HOST); for example, (LOGIN 'SUNRISE). You will then be prompted for a user name and password. This will be sent to the host when you attempt to CONNect or Chat.

Use the command CONN {HOST}<DIRECTORY>SUBDIR> to connect to a particular host. The local directory delimiters < and > can be used when connecting or file transferring. When communicating with a remote host you can specify the directory path as <DIRECTORY> SUBDIRECTORY> SUBDIRECTORY... , and the appropriate delimiters are presented to the remote host. Determination of what delimiter is presented depends upon the value of the OSTYPE field in the HOSTS.TXT file. If the field is empty, OSTYPE = 'Interlisp' is the default.

Using the above HOST.TXT file as an example you can do the following:

```
          UNIX  CONN {SUNRISE}<DIR>SUBDIR>SUBDIR>
           VMS  CONN {OBERON}<DIR>SUBDIR>SUBDIR>
       TOPS-20  CONN {RUTGERS}<DIR>SUBDIR>
SYMBOLICS-3600  CONN {SIMBA}<DIR>SUBDIR>
```

You can then do a DIR of the remote host, COPYFILE files to and from the host, assuming TCPFTP is loaded, MAKEFILE, etc.

Since the TCPFTP specification does not specify file type conventions, the variable TCP.DEFAULT.FILETYPES is used to associate a file's extension with the type of file it is. It is a list in the form (extension . type); for example,

```
((LCOM . BINARY)  (TXT . TEXT) etc)
```

Since Unix systems are case-sensitive, you should also have the lower case version of the file extensions on this list. If a file extension is not found on this list, the variable TCP.DEFAULTFILETYPE is used as the default file type during file transfers.

To Chat to a remote host, select Chat from the background menu and enter the host name when prompted. You will be prompted

for a Chat window and should then be able to chat to the host. If you have problems opening the Chat connection, try (CHAT 'HOST 'NONE). This will suppress the automatic login.

Making a Sysout that Contains TCP-IP

1. Load Medley sysout.
2. Create TCP host table.
3. Load TCPCONFIG.LCOM and run TCP.CONFIGURE if there is no IP.INIT file on local disk.
4. Load TCP.LCOM.TCPFTP.LCOM, TCPCHAT.LCOM.
5. Load TCPDEBUG.LCOM if you always want to have trace and echo facilities available.
6. Evaluate (TCP.STOP)
7. Evaluate (STOPIP)
8. Evaluate

   (SETQ RESTARTETHERFNS (LIST '(LAMBDA NIL (AND \IPFLG (\IPINIT)))))
9. Load any other files that you want in this sysout.
10. Evaluate SYSOUT to the device of your choice. Evaluate (\TCP.INIT) to re-enable TCP.
11. Load TCP sysout on other machine.
12. Create TCP host table.
13. Evaluate (TCP.CONFIGURE) and identify the new machine.
14. Evaluate (\TCP.INIT) to re-enable TCP.
15. Evaluate (\IPINIT) to restart the IPLISTENER process.

TCP-IP Protocol Layers

The TCP-IP family consists of four principal protocol layers: the link layer, the internet layer, the transport layer, and the application layer.

Link Layer

The physical link layer, the medium for transferring packets between hosts, is assumed to be any medium capable of transporting packets of data between hosts. Common link layers in this family include the Ethernet and the ARPANET.

The Address Resolution (AR) Protocol enables hosts to map between internet addresses and link layer addresses.

For example, the internet layer protocol IP (see below) uses a 32-bit combined unique host and network address; the host

address field is of variable size and depends on the pattern encoded in the high-order bits of the address. On the other hand, the 10 MB Ethernet uses a fixed-size 48-bit unique host address. The Address Resolution protocol, documented in RFC826, allows hosts to discover dynamically the link layer address equivalents of other internet hosts.

Internet Layer

The internet layer is responsible for routing packets between hosts. Unlike the link layer, the internet layer is capable of moving packets between hosts that are not connected to the same network. The term IP in TCP-IP refers to the Internet Protocol, the protocol that performs this task in the TCP-IP family. IP is documented in RFC791. IP is not assumed to be error-free; packets may be lost or duplicated while moving through the internet. It is the responsibility of the transport layer (see below) to guarantee perfect delivery, should the client require it.

IP also depends on an associated protocol called the Internet Control Message Protocol (ICMP). ICMP is responsible for handling exception conditions that arise between hosts using IP. Such conditions include the inability to deliver packets, errors in packet formats, etc. ICMP is documented in RFC792.

Transport Layer

The transport layer is responsible for assuring error-free, duplicate-free, sequenced delivery of packets between communicating processes. The most common transport layer is TCP, the Transport Control Protocol. TCP maintains the appearance of a perfect byte stream between processes. TCP is documented in RFC793.

An unreliable transport layer called the User Datagram Protocol (UDP) allows for packet exchange between communicating processes, but makes no attempt to guarantee delivery, suppress duplication, etc. Clients of UDP must provide their own error-recovery mechanisms if necessary. UDP is documented in RFC768.

Application Layer

Many applications exist in the TCP-IP family. The most common applications are file transfer, virtual terminal interaction, and mail delivery.

File Transfer

Two principal file transfer applications are in use: FTP, based on TCP and documented in RFC765; and TFTP (the Trivial File Transfer Protocol), based on UDP and documented in RFC783. Both are implemented in Interlisp, and are discussed at greater length below.

Virtual Terminal Interaction

The TELNET protocol, documented in RFC854, specifies the protocol for virtual terminal interaction between a user and a remote system. The Chat module will use the TELNET protocol to connect to TCP-only hosts.

Mail Delivery

The Simple Mail Transfer Protocol (SMTP) enables the delivery of mail between system elements using TCP. It is not currently implemented in Interlisp. SMTP is documented in RFC821. The format of messages is described in RFC822.

A Primer on IP Networks

The Internet Protocol internetwork is a collection of IP networks, a subset of which may communicate with each other. Each network is assigned an IP address, which is composed of a network number and a host number. No two hosts in the internetwork have the same network and host number combination; the composition of the network and host number for a particular host unambiguously identifies that host within the internetwork.

Network Addresses

The address space of the internetwork is formed of the concatenated network and host numbers of its constituent hosts, and is 32 bits long. This 32-bit address space is currently partitioned into three classes of network addresses, known as class-A, class-B, and class-C:

Class-A addresses consist of 7 bits of network number and 24 bits of host number.

Class-B addresses consist of 14 bits of network number and 16 bits of host number.

Class-C addresses consist of 21 bits of network number and 8 bits of host number.

Thus, there may be 128 class-A networks, 16,384 class-B networks, and over two million class-C networks. In addition, a single class-A network has the capacity to address over 16 million hosts, while a class-C network can address only 255 hosts. The class to which a particular IP network belongs may be determined by examining the most significant bits of its address.

Network number assignments are strictly controlled by a central authority. Institutions requesting network assignments are given class-A, -B, or -C networks depending on their estimated eventual size (numbers of hosts). Sites without assigned network numbers may request an assigned number by contacting:

Joyce Reynolds
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90292-6695
Phone: (213) 822-1511
ARPANET: JKREYNOLDS@USC-ISIF.ARPA

IP addresses are normally stored or exchanged as single 32-bit numbers. The printed representation of an IP address takes the form W.X.Y.Z, where W through Z are the decimal equivalents of each of the 8--bit bytes that constitute the address. Class-A addresses are of the form N.H.H.H; class-B addresses are of the form N.N.H.H; and class-C addresses are of the form N.N.N.H, where N indicates a byte of the network number, and H indicates a byte of the host number.

Class-A: N.H.H.H The first number is between 0-127 (for example, 122.0.2.1)

Class-B: N.N.H.H The first number is between 128-191 (for example, 153.4.23.5)

Class-C: N.N.N.H The first number is between 192-255 (for example, 194.5.67.3)

For example, 36.47.0.12 is an address on network 36, a class-A network; and 192.10.200.1 is an address on network 192.10.200, a class-C address.

Broadcast Address

The Internet Protocol defines an address in which the host field contains all ones to be a broadcast address for its network. Thus, the address 36.255.255.255 is the broadcast address on network 36, and 192.10.200.255 is the broadcast address on network 192.10.200.

Subnets

It is quite common for class-B networks to be partitioned into a set of smaller subnetworks, which are really class-C networks, but have the wrong network number to be recognized as class-C networks. This is just as common is partitioning a class-A network into many class-B subnetworks. An implementation of TCP-IP that is not prepared to handle this violation of the IP standard will not be able to communicate with hosts on the same network but different subnetworks. Fortunately, extending an IP implementation to support subnetworks is straightforward.

SUBNETMASK is a 32-bit parameter that resembles an IP address. The purpose of the mask is to enable a host to determine when a destination IP address is or is not on the same subnet as the sending host itself.

The SUBNETMASK has the following properties:

- The bitwise-AND of a source host's address (for example, this machine) and the SUBNETMASK must be equal to the bitwise-AND of a destination host's address and the

SUBNETMASK if and only if the two hosts are on the same subnetwork.

- The bitwise-AND of a source host's address and the SUBNETMASK must not be equal to the bitwise-AND of a destination host's address and the SUBNETMASK if and only if the two hosts are on different subnetworks.

As an example, consider network 39.0.0.0. This is a class-A network. Suppose this network consists of a number of subnetworks; for example, subnetworks with numbers like 39.47.*.* and 39.9.*.*. According to the IP specification, these subnetworks should really be one monolithic network, such that a host desiring to communicate with any other host whose address begins with 39... should have to take no special action with regard to routing packets to that host. Let us assume that this is not the case. The only way a machine has of telling which hosts are on different networks is to compare the masked version of the address with the masked version of its own address.

To continue the example further, assume the following:

Host A has address 39.9.0.6. Host A's SUBNETMASK is 39.255.0.0.

Host B has address 39.9.0.7. Host B's SUBNETMASK is also 39.255.0.0.

Host C has address 39.47.0.6. Host C's SUBNETMASK is also 39.255.0.0.

When host A sends to host B, it compares its masked address with host B's masked address, and finds them equal:

39.9.0.6 AND 39.255.0.0 = 39.9.0.0; 39.9.0.7 AND 39.255.0.0 = 39.9.0.0

However, when host A sends to host C, it finds the masked comparison does not match:

39.9.0.6 AND 39.255.0.0 = 39.9.0.0; 39.47.0.6 AND 39.255.0.0 = 36.47.0.0

Class-A networks that are subdivided into class-B subnetworks have SUBNETMASKs that look like X.255.0.0, where X is the class-A network number. Likewise, class-B networks subdivided into class-C subnetworks have SUBNETMASKs that look like X.Y.255.0, where X.Y is the class-B network number. Finally, networks in which subnet routing is not in use have SUBNETMASKs identical to their network addresses. For example, if network 36 did not use subnet routing, its SUBNETMASK would be 36.0.0.0.

The definitive document on this approach to subnetwork routing is RFC940.

Interlisp Files

The files that implement the TCP-IP protocol suite are divided into two classes: those that implement low-level functionality, normally not of interest to general users, and those that implement higher-level functionality for user programs (either application or transport layer protocols).

The higher-level functions reside in the files TCP, TCPDEBUG, TCPFTP, TCPCHAT, TCPNAMES, TCPPUDP, and TCPFTP.

TCP — The TCP layer. Implements TCP streams, based on the buffered TCP device (for example , BIN runs in microcode).

TCPDEBUG — Contains routines to help debug TCP and TCP-based applications.

TCPFTP — Contains the TCP-based file transfer protocol. Creates a new virtual I/O device, allowing transparent filing operations with TCP-only hosts.

TCPFTPSRV — Contains the TCP-based FTP server program. When the server program is running on a Xerox 1100-series workstation, other TCP-based hosts may transfer files to and from the workstation.

TCPNAMES — Implements translation of file name formats between operating system types.

TCPCHAT — Implements the TELNET protocol for the Chat system.

TCPUDP — Contains the UDP layer.

TCPTFTP — Implements the TFTP protocol. Creates a buffered TFTP device to allow efficient bulk transfer between hosts.

The low-level functions reside in the files TCPLLIP, TCPLLICMP, TCPLLAR, TCPHTE, and TCPCONFIG.

TCPLLIP — Implements the IP layer.

TCPLLICMP — Implements ICMP for IP.

TCPLLAR — Implements AR for the 3- and 10-megabyte Ethernets.

TCPHTE — Implements the functionality necessary to parse RFC810-style HOSTS.TXT files. This allows name-to-address translation within the Interlisp host.

TCPCONFIG — Provides a function to carry on a configuration dialog when TCP-IP is first installed on a machine. This file needs to be loaded only once, to produce the file {DSK}IP.INIT. Thereafter, TCPCONFIG is needed only to reestablish or modify IP parameters.

TCP

TCP implements the transport control protocol for Interlisp. After TCP is loaded, Interlisp supports a TCP stream capable of bidirectional I/O to a remote system element. The following functions are intended for use by applications programs.

(TCP.OPEN *DST.HOST DST.PORT SRC.PORT MODE ACCESS NOERRORFLG OPTIONS*) [Function]

Opens a TCP stream to *DST.PORT* on *DST.HOST* from *SRC.PORT*.

DST.HOST can be a host name, an IP host address in text format (such as 192.10.200.1), or the 32-bit integer representation of an IP host address as returned by the function DODIP.HOSTP (which is documented under TCPLLIP).

DST.PORT is a 16-bit number representing a TCP port open in LISTENING mode on the remote system.

SRC.PORT is also a 16-bit number, but may be supplied as NIL to obtain a defaulted unique local port number.

MODE is either ACTIVE, meaning to act as initiator of the connection, or PASSIVE, meaning to wait for a remote system element to initiate the connection.

ACCESS is either INPUT, OUTPUT, or APPEND (OUTPUT and APPEND are treated in the same manner).

If *NOERRORFLG* is non-NIL, TCP.OPEN will return NIL if the connection fails; otherwise, TCP.OPEN will call ERROR to signal failure.

OPTIONS is an optional parameter which allows the application program to control some of the characteristics of the TCP connection. *OPTIONS* is supplied in property-list format. Currently, the only recognized option is MAXSEG, whose value should be the number of data bytes the remote TCP sender is allowed to place into a single TCP segment (Ethernet packet). The maximum value of MAXSEG is 536.

If TCP.OPEN succeeds, it returns a STREAM open as specified by *ACCESS*. The generic operations BIN, BOUT, PEEKBIN, BINS, BOUTS, READP, EOFP, OPENP, GETFILEPTR, FORCEOUTPUT, and CLOSEF may be performed on streams opened for suitable access.

(TCP.OTHER.STREAM *STREAM*) [Function]

Returns the STREAM open in the other direction with respect to *STREAM* (for example, if *STREAM* is open for INPUT, TCP.OTHER.STREAM returns a STREAM open for OUTPUT, and vice versa).

(TCP.URGENT.EVENT *STREAM*) [Function]

Returns an event upon which a user process may wait for URGENT data to arrive on *STREAM*.

(TCP.URGENTP *STREAM*) [Function]

Returns T if *STREAM* is currently reading URGENT data.

(TCP.URGENT.MARK *STREAM*) [Function]

Marks the current point in *STREAM* as the end of URGENT data. *STREAM* must be open for OUTPUT.

(TCP.CLOSE.SENDER *STREAM*) [Function]

Closes the output side of *STREAM*, which may be either the INPUT or OUTPUT stream for the connection. This function differs from CLOSEF in that the INPUT side of the connection is not closed (although the remote system element may close the connection once the local output side of the connection is closed).

(TCP.STOP) [Function]

Disables the TCP protocol, closing all open TCP streams.

(\TCP.INIT) [Function]

(Re)initializes the TCP module.

\TCP.DEFAULT.RECEIVE.WINDOW [Variable]

Is the default number of bytes allowed outstanding from the remote system. It is initially 4,096.

\TCP.DEFAULT.USER.TIMEOUT [Variable]

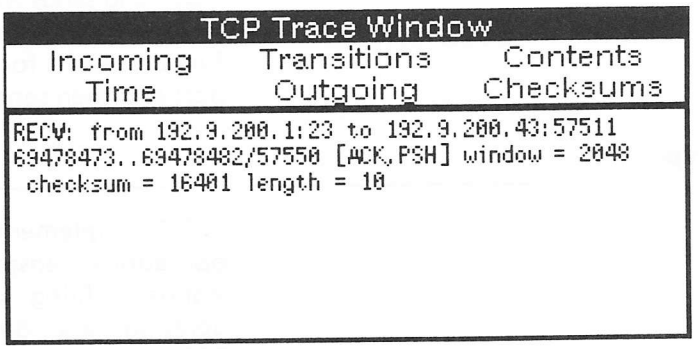
Is the default number of milliseconds a remote system element is allowed to remain silent before the TCP connection is declared broken. It is initially 60,000.

TCPDEBUG

TCPDEBUG implements tracing and test functions used to debug TCP and TCP-based applications.

(TCPTRACE) [Function]

Opens a trace window and attaches a menu to the window's top.



The menu entries represent state changes or data elements to be traced; each entry is a toggle. Clicking on the toggle once will activate the trace of the particular element and will gray-over the entry; clicking a second time will deactivate the tracing and ungray the menu item. The following data elements/transitions may be displayed:

Contents	Displays a line's worth of packet contents. The Incoming or Outgoing switch must be on.
Incoming	Displays incoming data.
Outgoing	Displays outgoing data.

Checksums	Displays checksums for each TCP segment.
Time	Displays the time interval since the last action on the connection.
Transitions	Displays state transitions on the TCP state machine.

(PPTCB *TCB FILE*) [Function]

Prints the state of a TCP connection. PPTCB is normally the INFO function for the process that monitors a connection; thus, selecting INFO in the process status window will cause a window to pop up containing a report on the status of the associated connection.

(TCP.ECHOTEST *HOST NLINES*) [Function]

Opens a TCP connection to the TCP echo port on *HOST* and sends *NLINES* of random text. The echo responses are displayed in a window. If *NLINES* is NIL, the echo test will run forever.

(TCP.ECHO.SERVER *PORT*) [Function]

Starts a TCP echo server on *PORT* (defaults to the TCP echo port). It is usually more useful to start the echo server as a process by doing (ADD.PROCESS '(TCP.ECHO.SERVER *PORT*)).

(TCP.SINK.SERVER *PORT*) [Function]

Starts a TCP sink server on *PORT* (defaults to the TCP sink port). Any data sent to this port will be acknowledged and discarded. As with the TCP echo server, it is usually more useful to start this server as an independent process.

(TCP.FAUCET *HOST PORT NLINES*) [Function]

If *HOST* is non-NIL, this function opens a connection to *PORT* on *HOST* and sends *NLINES* of text (the default is to send lines of text forever). *PORT* defaults to the TCP sink port. If *HOST* is NIL, this function waits for a remote system to connect to the TCP faucet port and then sends out *NLINES* of random text.

TCPFTP

TCPFTP implements a virtual I/O device that performs Lisp filing operations transparently using the RFC765 FTP protocol. The standard filing operations of reading, writing, renaming, deleting, and directory enumeration are supported by the TCPFTP device. However, neither random access filing nor GETFILEINFO are supported, as there is no protocol specification for performing these operations on files. Interlisp operations such as RECOMPILE will not work when files are stored on TCPFTP file servers.

Once TCPFTP is loaded, filing operations should be transparent to users; no additional initialization need be performed. There are, however, two important global variables:

TCPFTP.DEFAULT.FILETYPES [Variable]

This variable is an association list, keyed by common extensions of file names, and contains appropriate file types (for example, TEXT or BINARY) for such files. The TCPFTP protocol provides no

mechanism for determining the type of a file about to be retrieved. The file type is usually known in the case of output operations (for example, COPYFILE or MAKEFILE to a file server). However, in the case of COPYFILE from a file server, the TCPFTP module has to infer the file type from other knowledge. The module tries to match the extension of the file name with an entry on the list TCPFTP.DEFAULT.FILETYPES. If it finds a match, it uses the value of the entry in the list as the file type of the file; if it doesn't find a match, it uses the value of TCP.DEFAULTFILETYPE for the file type of the file.

TCP.DEFAULTFILETYPE [Variable]

If no matching extension is found for the file being opened, the TCPFTP module uses the value of TCP.DEFAULTFILETYPE as the file type of the remote file. The initial value of TCP.DEFAULTFILETYPE is BINARY; however, users may preset its value in their INIT.LISP files prior to loading TCP-IP.

The following functions are available for debugging broken file server connections.

(FTPDEBUG *FLG*) [Function]

If *FLG* is T, this function opens a scrolling trace window that displays FTP commands as they are issued. PUPFTP commands will also be displayed in this window (the window is the value of FTPDEBUGLOG).

(\TCP.BYE *HOST*) [Function]

Breaks an FTP connection to *HOST*.

(\TCPFTP.INIT) [Function]

(Re)initializes the TCPFTP module.

TCPFTPSRV

The TCPFTPSRV module contains a program which implements an FTP service for Interlisp. When this program is running on a workstation, other hosts are able to store and retrieve files from the workstation.

(TCPFTP.SERVER *PORT DEFAULT.FILE.PATH*) [Function]

To start the server program, evaluate the form (TCPFTP.SERVER). If *PORT* is supplied, the FTP server program will listen for connections on the TCP port specified by *PORT*; otherwise, the server will listen on the default FTP server port, port 21.

If *DEFAULT.FILE.PATH* is supplied, the initial path for resolving file names will be relative to *DEFAULT.FILE.PATH*; the default value of this variable is {DSK}<LISPFILES>.

TCPFTP.SERVER.USE.TOPS20.SYNTAX [Variable]

This variable controls whether file names sent back to FTP client programs are formatted in Tops-20 or Interlisp syntax. If the variable is true (the default), all file names will be formatted in Tops-20 syntax. This permits an Interlisp workstation to

masquerade as a Tops-20 mainframe for the purposes of file transfer to and from other vendors' machines.

TCPNAMES

The TCPNAMES module provides a set of functions for translating between the file-naming conventions of different operating systems. This is needed by the TCPFTP module in order for it to convert between Interlisp format file names and the file name formats of other operating systems.

(REPACKFILENAME.STRING *NAME FOROSTYPE*) [Function]

NAME is a file name in some operating system's format. *FOROSTYPE* is the name of an operating system. REPACKFILENAME.STRING attempts to translate *NAME* into a format acceptable to the operating system named by *FOROSTYPE*. *NAME* may be a string or atom; the function always returns a string.

Currently acceptable operating system types are:

IFS
INTERLISP
MS-DOS
SYMBOLICS-3600
TENEX
TOPS-20 (also TOPS20)
UNIX
VMS

(TI-Explorers should use TOPS-20 as their operating system.)

The correspondence between the target operating system type and the file name translation function is maintained in an extensible hash table.

(\REPACKFILENAME.NEW.TRANSLATION *OSTYPE FUNCTION*) [Function]

This function adds a new file name translation function for a new operating system type. The function must be a LAMBDA-NOSPREAD function, and must be prepared to receive either a single property-list format argument, such as would be returned by UNPACKFILENAME, or an arbitrary number of arguments in property-list format.

File names in the above format will be passed to the translation function adhering to the conventions of many operating systems; the function must recognize the operating system type and produce the desired output format, which must be a string.

\REPACKFILENAME.OSTYPE.TABLE [Variable]

This variable is the hash table that stores the correspondence between operating system types and translation functions.

TCPCHAT

TCPCHAT implements the TELNET protocol for virtual terminal I/O between Interlisp and a remote system. Once loaded into

Interlisp, the standard Chat system will use TCP TELNET to communicate with hosts that are believed to support the protocol.

No user-callable functions reside in this module, although the following variables may be of interest.

TCPCHAT.TELNET.TTY.TYPES [Variable]

This variable is an association list that maps internal names of Chat terminal emulators to official terminal names as specified in RFC884, the TELNET Terminal Type Option. This allows TCPCHAT to set the user's terminal type automatically when a connection is established.

TCPCHAT.TRACEFLG [Variable]

If this variable is non-NIL, TELNET negotiations will be printed to TCPCHAT.TRACEFILE (see below). This is sometimes useful in debugging negotiation problems.

TCPCHAT.TRACEFILE [Variable]

TELNET negotiations are printed to this file if TCPCHAT.TRACEFLG is non-NIL.

TCPUDP

UDP implements the user datagram protocol. The following functions are meant to be called by client applications.

(UDP.INIT) [Function]

Initializes the UDP module. This function is normally called when UDP is loaded and should not need to be called again under normal circumstances.

(UDP.STOP) [Function]

Disables the UDP module, closing any open UDP sockets.

(UDP.OPEN.SOCKET *SKT# IFCLASH*) [Function]

Opens a socket for UDP operations.

SKT#, if supplied, is a 16-bit number and will default to a number between 1,000 and 65,535.

IFCLASH specifies what to do if the requested socket is already open and is handled as in OPENPUPSOCKET and OPENNSOCKET (see the *IRM*).

It returns an instance of an IPSOCKET.

(UDP.CLOSE.SOCKET *IPSOCKET NOERRORFLG*) [Function]

Closes an open *IPSOCKET*. If *IPSOCKET* is not an open socket and *NOERRORFLG* is NIL, an error will occur; otherwise, NIL is returned if the socket is not active, and T is returned if the socket is active.

Any remaining packets on the socket's input queue are discarded when this function is called.

(UDP.SOCKET.EVENT *IPSOCKET*) [Function]

Returns an event that a process may use to wait for packet arrival on *IPSOCKET*.

(UDP.SOCKET.NUMBER *IPSOCKET*) [Function]

Returns the socket number of *IPSOCKET*.

(UDP.GET *IPSOCKET WAIT*) [Function]

Returns the next packet waiting on *IPSOCKET*. If no packets are waiting, does one of the following based on the value of *WAIT*.

NIL Returns immediately.

T Waits forever for a packet to arrive.

A *FIXP* waits up to *WAIT* milliseconds for a packet to arrive and returns NIL if none arrived during that time.

Thus, this function is like GETPUP and GETXIP.

(UDP.SETUP *UDP DESTHOST DESTSOCKET ID IPSOCKET REQUEUE*) [Function]

Initializes a fresh packet (as returned from \ALLOCATE.ETHERPACKET). The packet will be sent to *DESTSOCKET* on *DESTHOST*.

ID is a number to be placed in the IP header ID field (zero is fine).

REQUEUE specifies what to do with the packet after it is sent; NIL (the default) means no special treatment; FREE means to release the packet and return it to the free packet queue. Any instance of a SYSQUEUE will cause the packet to be queued on the tail of the specified queue.

UDP.SETUP initializes all IP and UDP fields and sets the packet up as a minimum-length UDP packet.

(UDP.SEND *IPSOCKET UDP*) [Function]

Sends *UDP*, a UDP-formatted packet, out from *IPSOCKET*.

(UDP.EXCHANGE *IPSOCKET OUTUDP TIMEOUT*) [Function]

Sends *OUTUDP* out from *IPSOCKET* and waits *TIMEOUT* milliseconds for a response; returns NIL if no response came in during the specified interval, or the packet that did come in during that time.

Clears the socket's input packet queue before waiting for a packet to arrive.

(UDP.APPEND.BYTE *UDP BYTE*) [Function]

Appends *BYTE* to the UDP data portion of *UDP* and increments the UDP and IP length fields by one.

(UDP.APPEND.WORD *UDP WORD*) [Function]

Appends *WORD* to the UDP data portion of *UDP* and increments the UDP and IP length fields by two.

(UDP.APPEND.CELL *UDP CELL*) [Function]

Appends *CELL* to the UDP data portion of *UDP* and increments the UDP and IP length fields by four.

(UDP.APPEND.STRING *UDP STRING*) [Function]

Appends *STRING* to the UDP data portion of *UDP* and increments the UDP and IP length fields by the length *STRING*.

TCPTFTP

TFTP implements the trivial file transfer protocol. This protocol is useful for transferring unimportant files rapidly (for example, between workstations and printers). The following user-callable functions exist.

(TFTP.PUT *FROM TO PARAMETERS*) [Function]

Sends a file to a TFTP host.

FROM may refer to any accessible file; *TO* must refer to a file accessible via TFTP.

No attempt is currently made to translate between Interlisp file name syntax and remote system file name syntax for *TO*.

For example, if *TO* resides on a Unix host, it would take a syntax like {HOST}/DIRECTORY/SUBDIRECTORY/FILENAME.

PARAMETERS is currently a list of parameters in the same format used by OPENFILE in .PARAMETERS; for example ((EOLCONVENTION 1) (TYPE TEXT)).

Note: TFTP transfers between Xerox Lisp and Unix hosts initiated from Xerox Lisp should have the PARAMETERS argument be '((EOLCONVENTION 10)).

(TFTP.GET *FROM TO PARAMETERS*) [Function]

Gets a file from a TFTP host. *FROM* must be a file accessible by TFTP; *TO* may be any file.

The file name syntax caveats for *FROM* are the same as for *TO* in TFTP.PUT. *PARAMETERS* is also as in TFTP.PUT.

(TFTP.SERVER *LOGSTREAM*) [Function]

Starts a TFTP server process.

LOGSTREAM may be left NIL, causing a new window to appear when the TFTP server is first invoked. Remote systems that support TFTP clients may store or retrieve files through any Interlisp workstation running the TFTP server.

The full Interlisp syntax for file names is supported; thus, requests to store files whose names include hosts will result in the Interlisp workstation's transparently storing the files on the designated hosts.

(\TFTP.OPENFILE *FILENAME ACCESS RECOG PARAMETERS*) [Function]

Returns a STREAM to open for *ACCESS* on *FILENAME*.

PARAMETERS is the usual format; TYPE is the only recognized parameter (BINARY opens a stream in *octet* format; TEXT, the default, opens a stream in NETASCII format; see RFC783).

BIN, BOUT, READP, EOFP, etc., may be used on this stream.

The stream is not RANDACCESSP.

(\TFTP.CLOSEFILE *STREAM*) [Function]

Closes the open stream. This is normally useful for streams open for OUTPUT; for INPUT streams, end-of-file will occur eventually.

TCPLLIP

For users planning implementations on top of IP, the following low-level TCP functions are available.

IP Socket Access

(\IPINIT) [Function]

Reinitializes the IP world; for example, after some catastrophe.

(STOPIP) [Function]

Disables IP.

(DODIP.HOSTP *NAME*) [Function]

If *NAME* is an integer, *NAME* is returned unaltered. If *NAME* is a text format IP host address (such as 192.10.200.1), DODIP.HOSTP returns its integer representation.

If *NAME* is a string or atom name, DODIP.HOSTP attempts to convert *NAME* to its IP host address integer value, using information supplied in the HOSTS.TXT file (see TCPHTE, below).

If *NAME* is unknown, DODIP.HOSTP returns NIL.

If *NAME* is known, it is cached with its corresponding address so that the function IPHOSTNAME may be used later to convert the address back to a name.

(IPHOSTNAME *IPADDRESS*) [Function]

Tries to convert *IPADDRESS* to a host name.

If *IPADDRESS* has no known name, it is converted to the text representation of an IP address (for example, 192.10.200.1).

(IPTRACE *MODE*) [Function]

Turns on tracing of IP activity. This function is like PUPTRACE and XIPTRACE, which are documented in the *IRM.*

If *MODE* is NIL, IP tracing is disabled.

If *MODE* is T, verbose IP tracing is enabled.

If *MODE* is PEEK, concise IP tracing is enabled. If *MODE* is either T or PEEK, the user is prompted for a window into which trace output will be printed.

(\IP.ADD.PROTOCOL *PROTOCOL SOCKETCOMPAREFN NOSOCKETFN INPUTFN ICMPFN*) [Function]

Defines a new IP-based protocol. The lowest-level IP functions maintain a list of active protocols and perform packet delivery based on the existence of open sockets for protocols of received packet types.

PROTOCOL is a protocol number, a number between 1 and 255. The following protocols are defined and should not be disturbed:

TCP	6
ICMP	1
UDP	17

SOCKETCOMPAREFN is a function with two arguments, an IP packet that has just been received and an open IPSOCKET. This function should return NIL if the packet does not belong to the supplied socket, or T if it does. The function will typically be interested in the IPSOCKET field of the IPSOCKET.

NOSOCKETFN is a function with one argument, an IP packet that has just been received. Its purpose is to handle received packets for which no socket can be found. If *NOSOCKETFN* is NIL, the default function, \IP.DEFAULT.NOSOCKETFN, will be used; this function simply returns an ICMP message indicating the socket is unreachable.

INPUTFN is a function with two arguments, a received IP packet and an open IPSOCKET. The *INPUTFN* is supposed to handle reception of packets when their destination socket has been found. If *INPUTFN* is NIL, the default function, \IP.DEFAULT.INPUTFN, will be supplied.

INPUTFN enqueues the received packet on the IPSQUEUE field of the IPSOCKET if the current queue length (stored in the IPSQUEUELENGTH field) is less than the allocated length (stored in the IPSQUEUEALLOC field).

I*NPUTFN* also increments the IPSQUEUELENGTH field, and notifies the event stored in the IPSEVENT field.

ICMPFN is a function with two arguments and is called when an ICMP packet referring to the protocol is received. The first argument is a pointer to the received ICMP packet. The second argument is a pointer that may be used as if pointed to the original outgoing packet included in the ICMP data. This allows the protocol functions to parse the data in the ICMP packet to determine which socket sent the offending packet. The *ICMPFN* must never attempt to deallocate the packet identified by the second argument; however, it is quite permissible (and expected) that the *ICMPFN* will release the packet identified by the first argument. The default *ICMPFN* simply releases the packet identified by the first argument.

\IP.ADD.PROTOCOL returns an IPSOCKET datum, which represents the active protocol; it is not in fact a useful IPSOCKET and may be safely ignored.

(\IP.DELETE.PROTOCOL *PROTOCOL*) [Function]

Deactivates a protocol with protocol number *PROTOCOL*. Any open sockets are closed.

(\IP.OPEN.SOCKET *PROTOCOL SOCKET NOERRORFLG SOCKETCOMPAREFN NOSOCKETFN INPUTFN*) [Function]

Attempts to open an IPSOCKET for protocol *PROTOCOL*.

SOCKET is the identifying information for this socket; this quantity will be EQUAL-compared with other sockets open on *PROTOCOL*. Should a match be found, an error will occur unless *NOERRORFLG* is T, in which case the existing socket will be returned.

SOCKETCOMPAREFN, *NOSOCKETFN*, and *INPUTFN* may be supplied to override the functions specified when the protocol was defined; they are not normally useful, however.

(\IP.CLOSE.SOCKET *SOCKET PROTOCOL NOERRORFLG*) [Function]

Closes a socket open on *PROTOCOL*. *SOCKET* is the same quantity passed to \IP.OPEN.SOCKET; it is currently not an instance of an IPSOCKET. If *NOERRORFLG* is T, an error will not occur if the socket is not found.

IP Packet Building

The following functions are useful for placing bytes into IP packets (as allocated by \ALLOCATE.ETHERPACKET).

Note that most applications will probably want to define a block record to overlay the data portion of an IP packet. Here is an example of such a block record.

Note: Users who are developing new IP-based protocols will need to load EXPORTS.ALL from the library.

```
(ACCESSFNS UDP
    ((UDPBASE (\IPDATABASE DATUM)))
    (BLOCKRECORD UDPBASE
        ((UDPSOURCEPORT WORD)
        (UDPDESTPORT WORD)
        (UDPLENGTH WORD)
        (UDPCHECKSUM WORD)))
    (ACCESSFNS UDP
        ((UDPCONTENTS
        (\ADDBASE
            (\IPDATABASE DATUM)
            (FOLDHI \UDPOVLEN BYTESPERWORD))))))
```

(\IP.APPEND.BYTE *IP BYTE INHEADER*) [Function]

Appends *BYTE* to the IP data portion of *IP* and increments the IP length field by one. If *INHEADER* is T, the IPHEADERLENGTH field is appropriately incremented so that the bytes appear to have been appended to the options portion of the IP header. There must not be any data bytes in the data portion of the packet if this function is to work correctly.

(\IP.APPEND.WORD *IP WORD INHEADER*) [Function]

Appends *WORD* to the IP data portion of *IP* and increments the IP length field by two. *INHEADER* is as in \IP.APPEND.BYTE.

(\IP.APPEND.CELL *IP CELL INHEADER*) [Function]

Appends *CELL* to the IP data portion of *IP* and increments the IP length field by four. *INHEADER* is as in \IP.APPEND.BYTE.

(\IP.APPEND.STRING *IP STRING*) [Function]

Appends *STRING* to the IP data portion of *IP* and increments the IP length field by the length *STRING*.

IP Packet Sending

(\IP.SETUPIP *IP DESTHOST ID SOCKET REQUEUE*) [Function]

Initializes *IP*. This function should be called just after *IP* is obtained from \ALLOCATE.ETHERPACKET; if this is not done, the append functions above will fail.

DESTHOST is the 32-bit IP address to which this packet will be sent.

ID is an arbitrary 16-bit quantity that will become the IPID field of the packet.

SOCKET is the open IPSOCKET from which the packet will be sent.

REQUEUE defaults to FREE and controls the disposition of the packet after transmission (see the *IRM* for the documentation of SETUPPUP or FILLINXIP).

(\IP.TRANSMIT *IP*) [Function]

Tries to send *IP*. Performs IP checksum algorithm prior to sending. Returns NIL if successful, otherwise it returns a status indication, such as NoRouting or AlreadyQueued. This function is like SENDPUP and SENDXIP, except that no socket argument is required.

TCPHTE

HTE provides functions for parsing HOSTS.TXT files as documented by RFC810. This file is loaded automatically by LLIP and is used by \IPINIT to read in the initial file, HOSTS.TXT. The following variable and function may be of interest.

HOSTS.TEXT.DIRECTORIES [Variable]

Is the search path for the file HOSTS.TXT. This variable is initialized to NIL; thus the search path to be used is by default DIRECTORIES.

(\HTE.READ.FILE *FILE WANTEDTYPES*) [Function]

Reads a HOSTS.TXT file.

WANTEDTYPES is a list of types drawn from the set *{HOST, NET, GATEWAY}*, to be read from the file; types not specified in *WANTEDTYPES* are ignored. *WANTEDTYPES* defaults to *(HOST)*.

TCP Debugging Aids

With TCPDEBUG loaded use (TCPTRACE T) to open up a trace window of TCP traffic. The appropriate items need to be selected from the windows menu in order for data to be seen.

(SETQ TCPCHAT.TRACEFLG T) will print TELNET negotiations to a file, which is what the variable TCPCHAT.TRACEFILE points to.

(FTPDEBUG T) opens a scrolling trace window that displays FTP commands as they are issued. You will see unencrypted passwords if they are issued.

Limitations

You must use the 1186 microcode in order for TCP-IP to work on an 1186 (microcode for the 1185 will not do).

TCP-IP will not work with Unix systems that have trailer encapsulation enabled. Connections will hang and then eventually break.

Directory enumeration on a VMS system results in NIL.

Known Problems in TCPFTPSRV

It does not handle error conditions in the middle of file transfers.

Doing a DIR gives you only filename and version: no author, creation date, etc. This is because the TCPFTP protocol specification doesn't support author, creation date, etc.

If there are multiple files on the system, deleting a file without specifing a specific version deletes the most recent version. The workaround is to give the specific version to delete.

The subdirectory structure is not presented back to the client host. If you have a file on both the <lispfiles> directory and a subdirectory, when you do a DIR *.* you do not see the subdirectory listed, but you do see that there are two files on the host with the same version number.

References

Users with access to the ARPANET may retrieve any RFC from host SRI-NIC.ARPA with the file transfer protocol (FTP) anonymous log-in option. RFCs are stored under <RFC>RFC*nnn*.TXT, where *nnn* is replaced by the number of the particular RFC.

From points on the Xerox internet, the RFC files can be retrieved from {Indigo}<RFC>. {Indigo} is an IFS host. From Lisp, you can simply (LOGIN) and supply your GV credentials if you haven't

already, open a FileBrowser on that directory, and retrieve the file to the local workstation environment.

The following RFCs are mentioned in this manual:

RFC765 (superceded by RFC 959)
RFC768
RFC783
RFC791
RFC792
RFC793
RFC810 (superceded by RFC 952)
RFC814
RFC821
RFC822
RFC826
RFC854
RFC894
RFC895
RFC903
RFC904
RFC940

[This page intentionally left blank]

TELERAID

TeleRaid is an interactive debugger which can be used either to examine, from one workstation, the state of another workstation's virtual memory, or to look inside a sysout file.

Requirements

REMOTEVMEM, READSYS, RDSYS, VMEM

Either:

Ethernet PUP connection between the two machines. The machine which is to be examined must be in the TeleRaid mode; i.e., the shape of its cursor must be the TeleRaid prompt. It must also have a PUP address.

Or:

A sysout file.

Installation

Load TELERAID.LCOM and the required .LCOM modules from the library.

User Interface

The standard use of TeleRaid is to debug a workstation that has stopped at a maintenance panel halt. Pressing the UNDO key when the machine is in this state transfers control to a small TeleRaid server that responds to simple commands over the network. While the TeleRaid server is running, the cursor changes to TELERAID. Also, on a 1108 workstation, the previous contents of the maintenance panel are restored.

On a 1108 workstation, the maintenance panel halt condition is indicated by a four-digit code that begins with a 9. On an 1186, the four-digit code is displayed at the cursor.

The term "debuggee" is used to denote the sysout file or machine running the TeleRaid server, i.e., the one being debugged, while "debugger" refers to the machine that is viewing the debuggee's virtual memory (usually by running TeleRaid).

Function

(TELERAID *HOST RAIDIX*) [Function]

Enters an interactive debugger viewing the virtual memory of *HOST*, which must denote a machine running a TeleRaid server. *HOST* is either a host name or a PUP address.

RAIDIX is an optional number denoting the radix in which values are printed and numbers are accepted as input; if not specified, it defaults to 8 (octal). The only other accepted value for *RAIDIX* at present is 16, for hexadecimal input and output.

If you don't know a machine's PUP name or address, you can find out by typing control-P on the debuggee: control-P changes the maintenance panel to show the machine's PUP host number in decimal radix. You can also find out your PUP address when Lisp is running (rather than in a maintenance panel halt) by evaluating (PORTSTRING (ETHERHOSTNUMBER)). Users typically do this once and tape a note to the terminal so as to have this information handy.

If the debugger is on the same physical Ethernet as the debuggee, you can use that PUP host number directly as the *HOST* argument. Otherwise, you must convert the PUP host number to octal and use the general form of a PUP address, which is a string of the form *"net#host#"*.

For example, (TELERAID 12) debugs the machine whose PUP address is 12 decimal on the same network. (TELERAID "13#14#") debugs host 14 octal (12 decimal) on network 13 octal.

Note: If the control-P command displays zero in the maintenance panel, it means the machine does not have a PUP host number assigned, or the halt occurred so quickly after booting that the Ethernet has not been fully initialized. In this case, TeleRaid cannot be used. See the description of READSYS (below) for directions on TeleRaiding a sysout file.

TeleRaid Commands

Each TeleRaid command is a single character, followed by arguments appropriate to the command. In the description of the commands that follows, unless otherwise specified, numbers are assumed to be typed in the default radix (octal unless you have specified a different *RAIDIX* in the call to TELERAID).

Displaying a Stack

For casual users, the L command followed by several F commands generally provide the most useful information. Many of the other commands require some knowledge of the internal

representation of Lisp objects and stack frames, something that this document does not attempt to provide.

L — shows the stack of the debuggee, as a back trace consisting of a numbered sequence of frame names. The first frame is usually \MP.ERROR if you got here by a maintenance panel halt.

In the case of MP code 9305, the stack shown is the page fault handler's and is uninteresting, except for the argument to the \INVALIDADDR frame.

Use the control-L *P* command to see the stack of the process that took the fault.

control-L *type* — Shows the stack of the debuggee starting at some other place. The argument *type* is a single letter denoting which stack to view. The system has a number of special contexts, which are areas of stack space used by certain system routines.

Legal values of *type* are P (page fault), G (garbage collector), K (keyboard handler), H (hard return), S (stack manipulator), R (reset), and M (miscellaneous).

The most interesting of these for most users is P, which for MP code 9305 shows the stack in which the address fault occurred. In addition, *type* F lets you view the stack starting at an arbitrary stack frame; follow F with an octal number denoting the frame (as in the control-X command, below).

K *type* — Changes the type of stack link that the L and control-L commands follow to be *type*, which is either A or C. The default is to follow CLinks (control links). ALinks follow the chain of free variable access instead.

Viewing Frames From a Stack

After displaying a particular stack with the L or control-L commands, the following commands view individual frames from that stack:

F *number* — Prints the contents of frame *number*, where *number* is the number next to the frame name in the back trace.

Note: Unlike most other commands, *number* is in decimal.

The frame is printed in two parts, a basic frame containing the function's arguments and a frame extension containing control information, the function's local (PROG) variables, and dynamic values. On the left side of the printout are the octal contents of each cell of the frame, with an interpretation, usually as a Lisp value, on the right.

line-feed or control-J — Shows the next frame (closer to the root of the stack). Same as F $n + 1$, where n is the number of the frame most recently viewed. Immediately after an L or control-L command, n is zero, so line-feed views the first frame.

↑ — Shows the previous frame. Same as F $n-1$.

D *symbol* — Shows the definition cell for *symbol*. A definition cell containing all zeros denotes an undefined function. A definition cell whose left half is less than 400 denotes an interpreted definition; you

can use the V command (below) to have it printed as a Lisp expression.

A *symbol* Shows the top-level value of *symbol*.

P *symbol* Shows the property list of *symbol*.

C *symbol* Prints (using PRINTCODE) the code definition for *symbol*.

V *hi lo* Interprets the virtual address *hi, lo* as a Lisp value and attempts to print it. Virtual addresses appearing in stack frames are already interpreted for you by the F command, as are those in value cells (the A command) and property lists (the P command), but you may want to use the V command if you find a virtual address inside some other structure.

B *hi lo count* Prints *count* words of the raw contents of the virtual memory starting at virtual address *hi, lo*. This is most useful for examining the contents of a datatype, which other commands simply print as its virtual address, i.e., in the form {type}#hi,lo.

_ *hi lo number* Sets the contents of the word at virtual address *hi, lo* to be *number*. This command obviously should be used with care.

control-V *symbol atomicValue* Sets the top-level value of symbol to be *atomicValue*, i.e., this is a remote SETTOPVAL. Only symbols and small integers are acceptable values to set. In addition, if the previous value was not a symbol or small integer, it is not reference counted correctly, so will not be garbage collected.

U Displays the debuggee's screen on your own (just the screen bit map, not the cursor). Typing any character restores your own screen. If the debugee's screen is larger than the debugger's, then you'll see that portion of the screen that fits. You can move the image of the remote screen by pressing the left mouse button and dragging the image, much like an over-size icon.

control-Y Enters the Old Interlisp Executive under TeleRaid, where you can evaluate arbitrary Lisp expressions or call some of the functions listed below to perform TeleRaid operations for which there is no command.

Use the Interlisp Executive's OK command to exit and return to TeleRaid.

Q Quits TeleRaid without affecting the debuggee.

control-N Executes the CONTROL-N TeleRaid command in the debuggee, i.e., causes the debuggee to resume execution, and quits TeleRaid. This command should not be used unless you are sure that the debuggee is resumable.

Viewing the System Stack

The following commands are for use by experts in stack format. A stack address is a number in the default radix denoting where the object of interest starts.

W *address* Walks sequentially through the system stack (i.e., by stack address, not by control or access links) starting at *address*, showing the stack frame type and its name (for frame extensions). If *address* is not given, this command shows the

entire user stack. For the READSYS function (see next section) the walk starts at zero, so it shows the system stack as well.

control-F *address* Prints the basic frame stored at *address*.

control-X *address* Prints the frame extension stored at *address*.

S *address count* Prints raw contents of the stack (as with the B command) starting at *address* for *count* words.

Functions for Saving Work

The following functions do not have corresponding TeleRaid commands, but may be useful to call in the executive obtained from the control-Y command. They can be used to try to patch a broken sysout back into shape, or at least to save some of the work out of a workstation in a maintenance panel halt. Further functions like these can be written using the functions described in the next section.

(VLOADFNS *FN*) [Function]

Reads the EXPR definition of *FN* from the remote environment and stores it locally on *FN*'s EXPR property. *FN* can be a single symbol or a list of symbols.

(VLOADVAR *VAR*) [Function]

Locally sets the variable *VAR* to be the remote top-level value of *VAR*.

(VSAVEWORK) [Function]

Attempts to figure out what has changed and not been saved in the remote environment by looking at CHANGEDFNSLST, CHANGEDVARSLST and the property lists of files on FILELST. For each changed function or variable, it asks you whether to save it, and if so, it uses VLOADFNS or VLOADVAR to fetch it. You can then save the functions or variables from the locally running Lisp.

VSAVEWORK does not know how to save records, properties, etc., although a knowledgeable programmer could use the functions described in the next section to extend VSAVEWORK.

(VUNSAVEDEF *FN*) [Function]

Attempts to do a remote UNSAVEDEF by going down the VGETPROPLIST of *FN*, looking for properties CODE, BROKEN, and ADVISED. If it finds one, it stores the corresponding code object in *FN*'s remote definition cell, and prints a message saying what it has done.

For example, if you've managed to break something that's used by the interpreter, and have thus gotten into a recursive break, you might be able to recover by VUNSAVEDEFing it, then doing a control-D on the remote machine.

(VYANKDEF *NEWSYMBOL OLDSYMBOL*) [Function]

Yanks the definition from function *OLDSYMBOL* and stores it into *NEWSYMBOL*. For example, (VYANKDEF 'PRINTBELLS 'NILL) turns off ringing of the bell in the remote environment.

Note: VUNSAVEDEF and VYANKDEF do not adjust reference counts, or interact correctly with BREAK and ADVISE. They should be thought of as emergency patches designed to get the system running long enough to save state and bail out. In particular, do not call UNBREAK or UNADVISE on a function that you have applied VUNSAVEDEF to, and do not alter or remove its CODE, BROKEN, or ADVISED property. Similarly, do not redefine the function *OLDSYMBOL* that you have yanked a definition from.

Implementation

TeleRaid is implemented in two parts: ReadSys, which reads a remote system's virtual memory, and VRaid, the interactive debugger described above. The remote virtual memory can be either a workstation running a TeleRaid server or a sysout file. The functions inside TeleRaid look like normal Lisp functions, but they are designed to operate on the remote virtual memory, rather than the normal (local) virtual memory. The remote versions of functions normally begin with V.

In general, TeleRaid is not a facility for the casual user. It is mostly used by system implementors performing very low-level debugging. The set of functions described here is a partial list, intended to help the serious programmer who has some interest in doing this kind of debugging.

Reading the Remote Vmem

(READSYS *FILE WRITEABLE*) [Function]

Opens the remote virtual memory *FILE*, which should be the name of a file in sysout format. If *WRITEABLE* is T, then the file is opened for write access, so that commands that alter the virtual memory (e.g., the __ and control-V commands) are permitted. The main use for this is to patch sysouts in simple ways (e.g., by changing a global flag from NIL to T).

FILE can also be a list of one element, the PUP address of a machine running a TeleRaid server, in the same form as the *HOST* argument to the function TELERAID. In this case, *WRITEABLE* is ignored.

If *FILE* is NIL, READSYS closes any open virtual memory file, clears its data structures and reverts to examining no virtual memory.

(VRAID *RAIDIX*) [Function]

Runs the TeleRaid interactive debugger on the virtual memory most recently opened by READSYS.

Manipulating the Remote VMem

The functions and macros described below directly manipulate the remote virtual memory. You can call them directly in the Lisp executive that you get by using the control-Y command under TeleRaid, or at the top level after calling READSYS. You can also, of course, write your own programs to use these functions. In order to use any of the macros below, you must LOADFROM the source file VMEM.

In the following functions, a pointer means a pointer into the remote virtual memory (the argument *PTR*), a 24-bit integer. All other arguments refer to local objects. Functions that fetch out of or store into the remote virtual memory operate on pointers. You can create a local copy of the structure denoted by a pointer by calling V\UNCOPY. You cannot do the inverse, i.e, create remote copies of local structures—the only local objects that you can translate into the remote virtual memory are symbols (assuming the same symbol exists remotely) and small integers.

Note: The functions that store into the remote memory should be used with care. None of these functions perform the proper reference counting. Therefore, if you are storing a value that ought to be reference-counted (roughly speaking, anything other than a symbol or small integer) and/or overwriting such a value, the garbage collector may get confused when the remote memory is resumed.

(VVAG2 *HI LO*) [Function]

Returns a pointer with hi-loc (top 8 bits) *HI* and lo-loc (low 16 bits) *LO*.

(VHILOC *PTR*) [Macro]

Returns the high part of *PTR*, i.e., (LRSH *PTR* 16).

(VLOLOC *PTR*) [Macro]

Returns the low part of *PTR*, i.e., (LOGAND *PTR* 177777Q).

(VADDBASE *PTR D*) [Macro]

Remote \ADDBASE: Returns a pointer that is *D* words beyond *PTR*, i.e., (IPLUS *PTR D*).

(V\UNCOPY *PTR*) [Function]

Returns a local copy of the remote structure pointed to by *PTR*. \UNCOPY only knows how to copy ordinary structures: symbols, integers (not bignums), floating-point numbers, characters, strings and lists. All other pointers, either as the argument to V\UNCOPY or inside structures copied by V\UNCOPY, are converted to local objects of type REMOTEPOINTER that print in the way that datatypes conventionally print—their contents are *not* copied.

(V\COPY *X*) [Function]

Returns a remote pointer to the local object *X*, which must be a symbol or small integer.

(VGETTOPVAL *ATOM*) [Function]

Returns a pointer to *ATOM*'s top-level value.

(VGETVAL *ATOM*) [Function]

Returns a local copy of *ATOM*'s top-level value, i.e., (V\UNCOPY (VGETTOPVAL *ATOM*)).

(VSETTOPVAL *ATOM VAL*) [Function]

Sets *ATOM*'s top-level value to be *VAL*, which must be a symbol or small integer.

(VGETPROPLIST *ATOM*) [Function]

Returns a pointer to ATOM's property list.

(VGETDEFN *ATOM*) [Function]

Returns a pointer to *ATOM*'s function definition.

(VTYPENAME *PTR*) [Function]

Returns the type name of *PTR*.

(VGETBASE0 *PTR*) [Function]

The most primitive fetching function: Returns the 16-bit integer stored in location *PTR*.

(VPUTBASE0 *PTR VAL*) [Function]

The most primitive storing function: Stores the 16-bit integer *VAL* into location *PTR*.

(VFIND.PACKAGE *NAME*) [Function]

Like the CL:FIND-PACKAGE, but returns the remote address of the named package or NIL if not found.

(VFIND.SYMBOL *NAME REMOTE-PACKAGE*) [Function]

Like the CL:FIND-SYMBOL, but returns the remote address of the named symbol.

(VGETBASE *PTR D*) [Macro]
(VPUTBASE *PTR D*) [Macro]
(VGETBASEBYTE *PTR D*) [Macro]
(VGETBASEPTR *PTR D*) [Macro]
(VPUTBASEPTR *PTR D VAL*) [Macro]

These are remote versions of \GETBASE, \PUTBASE, \GETBASEBYTE, \GETBASEPTR and \PUTBASEPTR, respectively. They are implemented in terms of VGETBASE0 and VPUTBASE0.

Limitations

TeleRaid uses PUP, thus the machine being examined must be on the same network or reachable via PUP gateways.

This code has one major shortcoming which will not normally turn up. If the local and remote sysouts conflict in their package setups, it is possible for this code to return symbols interned in what for the Teleraiding machine would be the correct package, but for the remote machine is in fact incorrect. The problem lies in the fact that you cannot uncopy a symbol correctly between two machines with incompatible package setups. An example of such a situation would be where on one machine the package FOO inherits BAR, and on the other BAR is present directly in FOO. BAR's package cell will be different in the two cases.

[This page intentionally left blank]

TEXEC

TExec is a version of the Interlisp-D executive which includes certain features of TEdit, so that commands can be edited, much like text. TExec preserves all of the functionality of the "old" executive (including history commands, ?=, DWIM, Programmer's Assistant, editing of the current input form, parenthesis matching/blinking, etc.) plus the ability to scroll anywhere in the output for viewing and/or copy-selecting old text.

TExec makes it easy to use Interlisp to get information, then use that information to build new commands to Interlisp.

TExec has two major advantages:

You can put into a window something longer than a windowful and still be able to scroll back and forth in it. In the regular exec window, all you see are the last few lines.

You can print something to the window, then use all or part if it at your next type-in.

Requirements

TEdit

Installation

Load TEDIT.LCOM and TEXEC.LCOM modules from the library.

User Interface

The Executive is described in the *IRM* and in the *Lisp Release Notes*.

TEdit is described in the *Lisp Documentation Tools* manual.

Starting TExec

TExec can be invoked interactively from the right-button (background) menu, or programmatically by calling

(TEXEC *REGION PROMPT MENUFN*) [Function]

If *REGION* is not specified, the system issues a prompt to create a window. If prompt is not supplied, a # is used as the prompt.

If *MENUFN* is not supplied, a command menu similar to TEdit is used. See the TEdit section in the *Lisp Documentation Tools* manual titled "Using the TEdit Window."

Differences between TExec and TEdit

The following TEdit commands are not included in the TExec main menu: LOOKS, SUBSTITUTE, QUIT, AND EXPANDED MENU.

TExec has two Find commands which are not in TEdit: FORWARD FIND and BACKWARD FIND.

FORWARD FIND searches forward from the beginning of the text stream if no previous text string has been found or if the caret is in the current/next type-in; otherwise the search continues forward from the last find.

BACKWARD FIND searces backwards from the type-in point if it is the first time, or from the last place it found the text. You can force BACKWARD FIND to start from the type-in point by placing the caret there with the mouse.

To allow the easy copy-selection of entire lines of input, use a carriage-return/line feed as the prompt, and the prompt will be printed on a different line from the type-in; e.g., (TEXEC REGION "<CR><LF>").

Pressing the escape key does not cause recognition of keywords in USERWORDS as it does under TTYIN. The ↑R (retype input) and case-changing commands of TTYIN are not implemented. Display stream graphics are not saved in the output.

Using TExec

TExec allows editing the current type-in using TEdit commands (see the TEdit section in the *Lisp Documentation Tools* manual titled "Editing Text"). Type-in is considered editable until a final matching right parenthesis, right bracket, or carriage return is typed, at which point it becomes immutable. Any output to a TExec window such as from CONTROL-T or ?= is placed in front of the current type-in so as not to interfere with your typing.

Unechoed input mode is implemented using a feature of TEdit known as invisible characters. Such characters, though invisible, are present in the buffer, and will be copied if they are within the bounds of a copy-selection. The primary terminal table, \PRIMTERMTABLE (the value of (GETTERMTABLE)) is used (different from TEdit) to allow control characters to be echoed as CONTROL-X (where is x is the control character), as they are in the Old Interlisp Executive.

The contents of a TExec window are saved in memory as a text stream. The maximum number of characters to be saved is specified by selecting the LIMIT command in the menu. When this limit is reached, characters are deleted from the beginning of the buffer as new ones are added to the end. The initial setting is 10,000 characters.

The escape key works the way it is described in the Programmer's Assistant section of the *IRM.* It is used as a character substitution mark by the Programmer's Assistant USE command.

Limitations

TExec does not understand Common Lisp syntax, so it is best to call it from an Interlisp exec.

= ? is not implemented.

[This page intentionally left blank]

TEXTMODULES converts source code files from File Manager format to Common Lisp style plain text and back again. When exporting to plain text, a small number of File Manager coms types are supported. When importing from a plain text file, several convenience features are available including comment upgrade and conversion of specific named defmacros into defdefiners.

NOTE: The Text File Translator changes source code format only; this is **not** an Interlisp to Common Lisp translator.

All symbols described in this section are in the TEXTMODULES package, nicknamed TM.

This section describes the load and make processes, and the static format of text files and their File Manager counterparts. The File Manager counterparts are discussed in increasing detail until their programmability is covered.

Overview

The Text File Translator supports the development of portable Common Lisp source code in the Lisp Environment. It brings portable Common Lisp sources into the File Manager without losing any of their contents. It also makes new text files based on the File Manager's "filecoms."

The original file's function and ordering are retained, but exact formatting is not. The pretty printer causes all comments and expressions on the text file to be uniformly formatted.

Exporting a source file into text and back again will lose grouping of definitions under their coms.

Installation

Load TEXTMODULES.DFASL from the library.

Dependencies

Special support for editing and printing of comments is required. This are provided by the SEDIT-COMMONLISP file. Some caveats on the editing of presentations are mentioned below.

The support file is automatically loaded by the TEXTMODULES file. SEDIT-COMMONLISP cannot operate without TEXTMODULES and must be loaded by it or after it.

File Manager source files created with **load-textmodule** depend on having TEXTMODULES and SEDIT-COMMONLISP loaded.

Programmer's Interface

load-textmodule *pathname* &key *module install package upgrade-comment-length join-comments convert-loaded-files defdefiner-macros*

[Function]

Like **lisp:load**; the file indicated by *pathname* is loaded, but in addition filecoms ar created and other information is stored for the File Manager. Key arguments are described below.

(See below under **Text File Format** for a description of the format of text files which can be read by **load-textmodule**).

Local bindings of reader affecting variables are established and set to Common Lisp defaults, except for the readtable.

A special readtable is used which creates internal representations for objects normally lost during reading (see below under **Presentation types**).

If there are some simple forms to set up the read environment at the front of the file, they are recognized and moved into a newly created makefile environment (see below under **Makefile Environment** for a complete description of this).

Each form is read from the file (one at a time). If the form is recognized a description of it is given to the File Manager and its definition is installed. If the form is not recognized it is wrapped in a "top level form" filecom and then installed by stripping presentation objects and evaluating.

defun in a **let** at top-level is treated like any top-level form. Such forms should be edited directly in the filecoms. Not doing this can have curious consequences, since calling **ed** on the function name will not modify the definition in the **let** (which remains in the FILECOMS as a top level form).

No forms after the read environment forms should change the reader's environment.

When the file has been completely read its content description is given to the File Manager. Also added to the content description are properties declaring its il:filetype as :compile-file and makefile-environment as that of the text file (whether given by setting forms at the front of the file or by default).

Several key options are available:

module A string or symbol used to create the symbol used as the File Manager's name of this module. Strings have their case preserved. Symbols have their name strings taken. Defaults to the uppercased root name of the path.

install T or NIL. Indicates whether the definitions in the file should be installed in the running system. Any package setup makes it mandatory to install the definitions in a source file; e. g. since :INSTALL NIL means forms in the file are not evaluated, any IN-PACKAGE form would not be evaluated and the file would be read in the wrong package. This can sometimes be worked around using the :PACKAGE argument.

package A package name or package, defaults to "USER". This is the package the file will be read into.

upgrade-comment-length A number or NIL. Defaults to the value of ***upgrade-comment-length*** (which defaults to 40). The length, in characters, at which single semicolon comments are upgraded to double semicolon comments.

join-comments T or NIL. Defaults to the value of ***join-comments*** (which defaults to T). Causes comments of the same level in the coms to be joined together. This makes for more efficient editor operation, but loses all formatting inside of comments; e.g. inter-comment line breaks are not preserved.

convert-loaded-files T, :QUERY or NIL. Defaults to the value of ***convert-loaded-files*** (which defaults to :QUERY). If a REQUIRE or LOAD statement is noticed at top level a recursive call to LOAD-TEXTMODULE will be made. With :QUERY turned on the user is first prompted. If the pathname specified in the LOAD or REQUIRE is computed based on variables in the file being loaded :INSTALL must be true. Complex systems that contain special loading functions will not be handled by this mechanism.

defdefiner-macros A list of defmacro names. Defaults to the value of ***defdefiner-macros*** (which defaults to NIL). If a top-level defmacro is found whose name is on this list, the defmacro will be translated into an IL:FUNCTIONS defdefiner form. The defdefiner form then creates a macro that builds definers. Definers are the basic definition units maintained by the File Manager. DEFUN is itself a defdefiner macro. A particular DEFUN form is a definer for the named function (see the Lisp Release Notes, 4. Changes to Interlisp-D in Lyric/Medley, Section 17.8.2 Defining New File Manager Types, for more information on the defdefiner form).

Warning: names on this list must be in the correct package, i.e. the one the file will be read in. A typical way to use this feature is:

- Examine the text source file for DEFMACRO forms that are used to create defdefiners.
- Make the package which the text file's IN-PACKAGE expression will later find.
- Do LOAD-TEXTMODULES giving the :DEFDEFINER-MACROS key argument a list of fully package qualified symbols naming the defdefiners contained in the file.

make-textmodule *module* &key *type pathname filecoms width* [Function]

The File Manager's description of the file *module* is used to create a text file. *module* may be provided as either a string or a

symbol. A string's case will be preserved. A symbol's name string is used. Keyword arguments are described below.

(See below under **File manager description of contents** for a description of filecoms that can be written out by **make-textmodule**.)

Local bindings of printer affecting variables are established and set to Common Lisp defaults, except for the readtable.

The file's environment is written out, based on its makefile-environment property (see below under **File manager source file format** for ways of expressing the environment).

The specially made description of the file's contents (from the File Manager) is iterated over to write out each form in the file.

Several key options are availible:

type A string, defaults to ".LISP". The file type extension to be used on the text file being written.

pathname A pathname, defaults to the module name merged with the extension and ***default-pathname-defaults***. The file which will contain the new text file.

filecoms A list of file commands may be supplied here. Defaults to the commands for the File Manager file named by module.

width A positive integer, defaults to 80. The width, in characters, of lines in the text file. Used by the prettyprinting routines for formatting.

Variables that control loading

join-comments [Variable]

T or NIL. Defaults to T. Causes comments of the same level in the file coms to be joined together. This makes for more efficient editor operation, but loses any formatting inside of comments, e.g. inter-comment line breaks are not preserved.

convert-loaded-files [Variable]

NIL, :QUERY or T. Defaults to :QUERY. Controls whether a LOAD or REQUIRE statement at top level in a loaded text file will cause the referred to file to be recursively load-textmodule'd. If the pathname specified in the LOAD or REQUIRE is computed based on variables in the file being loaded the :INSTALL argument to **load-textmodule** must be true.

upgrade-semicolon-comments [Variable]

NIL or a positive integer. Defaults to 40. Controls whether and at what length (in characters) a single semicolon comment is upgraded to a double semicolon comment. NIL inhibits upgrading.

defdefiner-macros [Variable]

A list of defmacro names. Defaults to NIL. If a top-level defmacro is found by LOAD-TEXTMODULES whose name is on this list, the defmacro will be translated into an IL:FUNCTIONS defdefiner form. The defdefiner form creates a macro that builds definers. Definers are the basic definition units maintained by the File Manager. DEFUN is itself a defdefiner macro. A particular DEFUN form is a definer for the named defun (see the Lisp Release Notes, 4. Changes to Interlisp-D in Lyric/Medley, Section 17.8.2 Defining New File Manager Types, for more information on the defdefiner form).

Warning: names on this list must be in the correct package, i.e. the one the file will be read in. A typical way to use this feature is:

- Examine the text source file for DEFMACRO forms that are used to create defdefiners.
- Make the package which the file's IN-PACKAGE expression will later find.
- Do LOAD-TEXTMODULES giving the :DEFDEFINER-MACROS keyword argument a list of fully package qualified symbols naming the defdefiners contained in the file.

Text file format

TextModules creates and understands the format of portable pure Common Lisp text files with very simple and constrained package setup information. The overall form of these files is described here as a guide to what sort of files may be imported.

An EMACS style mode line comment may optionally be present as the file's first item. It corresponds to the makefile-environment in the file manager.

```
; -*- Mode: LISP; Package: (FOO GLOBAL 1000); Base:10 -*-
```

mode For some versions of the EMACs editor this will declare the major mode, which arranges key to command bindings for LISP instead of documents.

package Name, used packages and initial space for symbols.

base Numeric "ibase"

The mode line is generated by TextModules and is provided purely as a convenience in transporting code to EMACS based environments. It has no effect on the File Manager. The makefile-environment is actually instated using expressions directly following the mode line.

The Common Lisp community has agreed that portable text files will use only one reader environment and hence not switch packages or alter the readtable partway through. TextModules

assumes that the reader environment is set up by the seven (plus two) standard environment modifying forms. These forms are recognized by the TextModules parser only if they appear at the front of the file and in order (comments being ignored):

Put **provide**
In **in-package**
Seven **shadow**
Extremely **export**
Random **require** (or **il:filesload**)
User **use-package**
Interface **import**
... **shadowing-import**
... **setf *read-base***
Commands Contents of module

Portable files may optionally add ***read-base*** setting and **shadowing-import** expressions. Also, **il:filesload** may be used in place of **require**, when a Lisp file (not containing a **provide** form) must be loaded.

Any one of these forms is optional, but they must appear first in the file (and in order) to be parsed into a makefile-environment when **load-textmodule** is called.

Warning: this software applies heuristics to the package forms to make them independent of the package environment they are read in. These may not work and it is recommended that the makefile-environment be checked for correctness after load-textmodule brings in the file. Complex package setups will almost certainly not be handled correctly and should be created in a separate file which the main text file can REQUIRE.

The contents of a text file are a sequence of forms. Certain forms are understood by the File Manager and hence specially recognized. These recognized forms include:

Comments which are translated into Interlisp style comments

Definers such as defun or defvar

eval-when which is translated into a File Manager eval-when

Read time conditionals which may become "unread objects"

All other forms are considered *top level forms* and simply saved as is.

Definers hold onto presentations, e.g., read time conditionals, as well as comments. Comments and presentations are always available to be edited.

To see if something is a definer form, examine the property list of its name (like **defun**). Use the Exec's **pl** (print property list) command to look for the property **:definer-for**.

Note that only the above kinds of forms will be recognized by the TextModules parser on a portable Common Lisp file.

There are a few somewhat common problems that can arise when importing a text file. Chief among these are "bootstrapping definitions" and circularity.

Problems to be aware of

Occasionally, when starting up a complex software system, it is useful to install a temporary definition until the mechanism required by the actual definition is in place. This can cause a definition by the same name to appear in more than one place in a text file. The Textmodules system will simply use the latter definition in the file, causing the next loading of it to fail for lack of the lost bootstrap definition. It is recommended that bootstrap definitions be made into "top level forms", e.g. a DEFUN can become a (SETF (SYMBOL-FUNCTION <name>) ...) form, DEFPARAMETER can become (SETF <name> ...), etc.

Also, some styles of programming may encourage creation of circular structure. Textmodules must map over top level forms to install presentations that contain comments, etc. Circular structures can cause these routines to fill memory with list structure.

File Manager source files

Unlike standard Common Lisp, the File Manager is designed to keep all of a file's contents resident in memory as structure, rather than text. This scheme allows very fast update and editing of definitions. To maintain its own source files the File Manager keeps descriptions of the format (makefile-environment) and contents (filecoms) of a file.

The makefile-environment of the file is used to note the readtable and package the rest of the file to be read and printed in, and any file dependencies.

The filecoms maintain a description of the top-level defining forms in a file, like function and variable definitions. They also store plain top-level forms. Within any form there can appear lisp data, like vectors or "number in a radix." How these are read and printed are controlled by presentation types (described separately; see below).

It is important to separate the environment of the file from its contents because the File Manager (not TextModules) first reads all the forms in the file, and then evaluates them. Text based source files sometimes change the package as needed. This cannot work for the File Manager since the file's forms are all read and then executed, i.e. the package changes would not occur until after the entire file had been read, and forms after any IN-PACKAGE form would have been read incorrectly.

The File Manager first reads the makefile-environment forms in a well known environment (INTERLISP package, INTERLISP readtable) evaluates them to find the environment of the rest of the source file, then reads the rest of the source file in that environment. This is why the package environment setup forms

are so carefully parsed out of text files being imported into the environment.

File Manager source files created by **load-textmodule** depend on both the TEXTMODULES & SEDIT-COMMONLISP modules. These must be loaded before source files created with them can be reloaded.

File Manager source file format

The makefile-environment of a "managed" source file is used to control both how the exported text file and managed source files are printed. It is kept in a property named **il:makefile-environment** on the symbol with the root name of the file (in the INTERLISP package). This property is automatically generated when a portable text file is imported. The property is itself a plist containing :readtable, :package and :base values. The readtable used is called "LISP-FILE", a readtable defined by the TextModules program (hence File Manager source files created by **load-textmodules** depend on TextModules). The part of the makefile-environment of main interest is the :package. It sets the package in which the exported text file is printed.

Three forms of the makefile-environment's :package property are recognized.

- a string or symbol naming a package
- a **defpackage** statement
- a **let** statement

A string or symbol is simply taken to name a package.

A **defpackage** statement will have its portable components translated into a **let** statement as described below.

A **let** used for the makefile-environment should bind ***package*** and contain some form of the standard seven package and module setup forms (See above under "Text file format"). It should finally return the altered value of *package*. For example:

```
(let ((*package* *package*))
  ...environment setup forms.
  *package*
)
```

The forms in this expression must be written in a standard, pre-existing package, such as USER or XCL-USER. This is to break the circularity of writing a package defining form in the package it defines.

The package defining expressions in the let should follow all of the rules for portable text files (See above under "Text file format"), e.g., they should appear in order.

File Manager description of contents

When a portable text file is imported its contents are parsed to produce the File Manager's filecoms (File Commands). File commands are a high-level way of viewing and controlling the ordering of definitions in a file. The following describes the filecoms produced when a text file is imported.

Forms on the file are either recognized as top level defining forms or wrapped in a "top level form" filecom. Several forms are recognized by TextModules itself and others can be added (see below under "Making new specifiers"). The constructs that recognize filecoms, both to export and import plain text, are called *specifiers*.

The following are placed in the filecoms based on the parsed contents of the text file:

(il:* *type string*) Contains a comment string. *type* is a symbol of one, two, three or four semicolons, or a vertical bar. This handles top level single, double or triple semicolon comments, as well as balanced comments. When viewed in SEdit these display in real comment format, instead of the internal list representation.

(eval-when *when* . *filecoms*) Wrapper with an evaluation time and containing more filecoms.

definers All definers are recognized, e.g. DEFUN, DEFMACRO, DEFVAR, DEFPARAMETER, DEFSTRUCT, etc. The definer specifier also converts DEFMACRO forms on the ***defdefiner-macros*** list to defdefiners during loading, and vice versa on printed to a text file.

(il:p (top-level-form *form*)) Top level form wrapper with a macro that calls the presentation translator. This filecom contains expressions which were not recognized and must be evaluated at load time. This kind of filecom also handles top level occurances of conditional read and read time evaluations (hash comma and hash dot). The top-level-form specifier also looks for LOADed or REQUIREd files and, depending on the variable ***convert-loaded-files***, attempts to convert the loaded files as well.

When the file is loaded and before evaluating these forms any presentation objects in them are stripped out (as for comments) or installed (as for read time evaluations). This is done by the TOP-LEVEL-FORM macro, which dispatches to the translation functions for the particular presentation objects. i.e., this allows comments to appear anywhere in the forms and not affect evaluation.

The above coms are created when a text file is imported. There are also a few specifiers provided to export filecoms, but not create them on import. These are convenient for exporting typical File Manager files. They are:

(il:coms . *filecoms*) Used to group together definitions in a File Manager source file. The *filecoms* are dumped onto the text file in order. No information is placed on the resulting text file to preserve the grouping of the filecoms; if the exported text file is later imported the coms grouping will not reappear.

(il:vars . *descriptors***)** As described in the *IRM*. The *descriptors* are exported as **defparameter** forms. If the exported text file is later imported these **vars** coms will reappear under **variables** coms.

(il:initvars . *descriptors***)** As described in the *IRM*. The *descriptors* are exported as **defvar** forms. If the exported text file is later imported these **initvars** coms will reappear under **variables** coms.

(il:constants . *descriptors***)** As described in the *IRM*. The *descriptors* are exported as **defconstant** forms. If the exported text file is later imported these **constants** coms will reappear under **variables** coms.

(il:props . *descriptors***)** As described in the *IRM*. The *descriptors* are exported as **(setf (getf ...) ...)** forms. If the exported text file is later imported these **props** coms will reappear under **p** coms (top-level forms).

(il:prop *props* . *symbols***)** As described in the *IRM*. The *props* and *symbols* descriptors are used to generate forms for export, e.g. **(setf (getf 'foo 'bar) 21)**. If the exported text file is later imported these **prop** coms will reappear under **p** coms (top-level forms).

(il:files . *items***)** As described in the *IRM*. The *items* are used to generate forms for export, e.g. **(load "Foo.lisp")**. All options except **noerror** are ignored, the latter will cause the **:if-does-not-exist nil** key argument to be included in the load expression. If the exported text file is later imported these **files** coms will reappear under **p** coms (top-level forms).

Making new specifiers

Specifiers are the glue that relate forms on a plain text file to filecoms in a File Manager source file. They can be considered addenda to the filepkgtype mechanism of the File Manager.

specifiers [Variable]

A list of specifiers (its default contents are described below). New specifiers should be added to this list. This list is searched linearly; its order is significant mostly in that the default top-level form recognizer must always be last.

make-specifier &key *name filecom-p form-p add-form install-form print-filecom*
[Function]

This function creates new specifiers for inclusion on the ***specifiers*** list. A specifier maps between the forms in a text file's contents and filecoms. It is the basis for importing and exporting top-level forms. Specifiers can be nested, as for EVAL-WHEN.

To do all of this a specifer contains functions that recognize forms of its kind on the text file and coms in filecoms, as well as functions that add the definition to the filecoms and install the definition as the one to be used at runtime. Finally, there is a function which prints a form onto a text file based on a com on the fileoms.

name A string naming the specifier.

filecom-p Predicate on FILECOM which returns true if it is the one used to represent this specifier in the filecoms of a managed file.

form-p Predicate on FORM, a form read from a text file being imported, which returns true if this is the specifier for the definition in FORM.

add-form Function of FORM and FILECOMS. FORM is a form read from a text file being imported, and which has already been confirmed with the *form-p* method. Should make the definition in FORM available (editable) in the programming environment (File Manager). It should make the definition editable and add a filecom for FORM to the FILECOMS description. It should return the new FILECOMS description. To *add-form* runs of subforms use **add-form** and **form-specifier** (see below).

Care should be used when making a definition editable. The simplest instance of this occurs when the FORM's definition is a definer. In this case its evaluation may be wrapped in a binding of **il:dfnflg** to **il:prop** to ensure that the definition form goes into the table of current definitions *without being evaluated*.

Adding a filecom to the FILECOMS should be done in a way that preserves ordering. The simplest way to do this is to append the new filecom to the end of the current FILECOMS.

install-form Function of a FORM which makes the definition in FORM the current one to be used in execution. If the defining mode flag indicates that the file is being loaded for editing only this function *will not be called* during loading of the form (il:dfnflg is set by the :install option to load-textmodule). To *install* runs of subforms use **install-form** and **form-specifier** (see below).

Care should be used when making a definition executable. The simplest instance of this occurs when the FORM's definition is a definer. In this case its evaluation may be wrapped in a binding of **il:dfnflg** to **t** to ensure that the definition form *is actually evaluated*.

print-filecom Function of FILECOM and STREAM which should generate a new line and pretty print a form, representing the FILECOM, onto the stream. To print runs of subforms use **print-filecom** and **filecom-specifier** (see below).

The semantics of the *add-form* and *install-form* methods remove some confusion between loading a definition into memory for editing (loading PROP) and installing that definition as the currently executable one (loading T). The *add-form* method makes the definition editable and the *install-form* makes it executable.

The following functions are used to handle subforms EG. In the eval-when specifier there are subforms that need to be parsed.

form-specifier *form* [Function]

Searches the current list of specifiers in an attempt to recognize *form* (as read from a text file being imported). Returns a

specifier for *form*. If none is found a warning is signalled and a "do nothing" specifier is returned.

filecom-specifier *filecom* [Function]

Searches the current list of specifiers in an attempt to recognize *filecom* (as found in the filecoms of the managed file). Returns a specifier for *filecom*. If none is found a warning is signalled and a "do nothing" specifier is returned.

add-form *form filecoms &optional specifier* [Function]

Adds the *form* to the *filecoms* description based on the add method in the *specifier*. If *specifier* is not provided **form-specifier** will be used to get it from *form*. Returns the new filecoms. **nil** is used as an empty contents description.

install-form *form &optional specifier* [Function]

If the current definition mode (**il:dfnflg**) allows it, installs the *form* as current and executable based on the *specifier*. If *specifier* is not provided **form-specifier** will be used to get it from *form*.

print-filecom *filecom stream* &optional *specifier* [Function]

Prints a new line on *stream* and then pretty prints a form representing the filecom onto the *stream*. If *specifier* is not provided **filecom-specifier** will be used to get it from *filecom*.

Presentation objects

Presentation objects represent things that normally disappear during reading, like comments or numbers written in a particular base. Each presentation object must be capable of being read from a text file, edited with SEdit, installed as it would be when normally read, and printed to a text file in its original form.

Presentations supported by Lisp

Many presentations are already supported by SEdit :

#\character	Character object.
#:symbol	Uninterned symbol.
#'function	Hash quote function abbreviation.
; comment	
;; comment	
;;; comment	
;;;; comment	Semicolon comments. Internal formatting is preserved when these are imported, including CRs and tabs, etc. Adjacent comments are *not* smashed together so that line breaks are preserved. A single leading space in a comment is ignored, since comments are always printed with a single leading space.

These comment types are represented internally in the same way as in Lisp, i.e., a list beginning with the symbol IL:*, following by

a symbol (interned in the INTERLISP package) containing one through four semicolons.

#|comment|# A balanced comment. Imported and exported by TextModules. Directly supported by SEdit as though it were a "level 5" semicolon comment.

This comment type is represented internally in a manner similar to semicolon comments, but where the symbol containing semicolons is replaced by a symbol whose name is the vertical bar character.

Presentations supported by SEDIT-COMMONLISP

Several presentations are specially supported by SEDIT-COMMONLISP. Any of these can be created using the following commands in SEdit:

Read time conditionals	Control-N Hash minus
	Control-P Hash plus
Read time evaluation	Control-Q Hash dot
Load time evaluation	Control-F Hash comma
Octal notation	Control-I Hash "O"
Hexidecimal notation	Control-J Hash "X"
Binary notation	Control-K Hash "B"

 # + feature form
#-feature form Read time conditionals.

A conditional expression can be either *unread* or *read* depending on whether the truth of its features expression and sign parse true (see *Common Lisp, the Language*). *Read* conditional expressions are stored as structure. *Unread* conditional expressions are stored as strings due to the potential inclusion of, e.g., numbers of higher precision, symbols in unknown packages, or the inclusion of unknown reader macros.

Unread read time conditionals are read by remembering file position, doing a read suppress read, backing up to the original position and saving all the characters between in a string. This means that streams from which conditional read presentations are read must be capable of random access (the TTY is not).

These are represented by hash-plus and hash-minus structures.

Editing of read time conditional presentations is not quite WYSIWYG. Feature symbols should always be given as keywords (this is done implicitly by the lisp reader, but not in SEdit) and unread forms appear in strings and must be edited as such.

#.form Read time evaluation. Hash dot is represented by the hash-dot presentation.

#,form Load time evaluation. Hash comma is represented by the hash-comma structure.

#Orational	
#Xrational	
#Brational	Rational representations (**O**ctal, he**X**idecimal, and **B**inary). These are represented by the hash-o, hash-x and hash-b structures.

Presentations not directly supported

It is not possible to directly edit the following presentations in SEdit:

#(contents)	Vectors
#rankA(contents)	Arrays
#S(name field1 value1 ...)	Structures
#*1010101	Bit vectors

Any of these may be edited by opening an inspector from SEdit: selecting the object and using the Meta-E command on it. Any of these may be created in SEdit by inserting the appropriate **make-** expression, selecting it, and using the Meta-Z command with **cl:eval** as the mutating function.

Presentations not supported

The following standard Common Lisp presentations are not supported by either SEdit or TextModules:

#n = object and ***#n#***	Object tag and reference notation
#baseRnumber	Radix notation

VIRTUAL KEYBOARDS

VirtualKeyboards lets you change the behavior of your Lisp workstation keyboard to mimic another keyboard, hence making yours a "virtual" version of that other keyboard. It also lets you display pictures of keyboards on your screen and use them as menus for typing occasional special characters. Several keyboards may be displayed on the screen at once, letting you switch easily among keyboards for several languages and making hundreds of characters available for typing.

The virtual keyboards supplied with the module are Dvorak, German, Greek, Italian, logic, math, Spanish, European accents, and standard Russian. You can also define new keyboards with the associated Keyboard Editor module, which lets you edit a keyboard while seeing the actual look of the characters.

The virtual keyboards can be used with TEdit, DEdit, in the Lisp Executive window, for any application with which you use your keyboard.

Requirements

You need one of the following files, as appropriate for the machine you are using:

DANDELIONKEYBOARDS
DORADOKEYBOARDS
DOVEKEYBOARDS

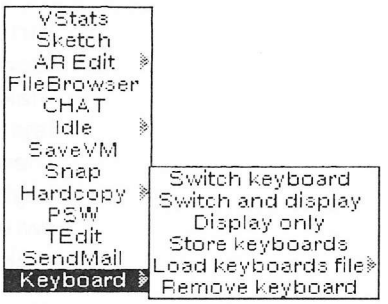
Installation

Load VIRTUALKEYBOARDS.LCOM from the library. VirtualKeyboards loads the xxxKEYBOARDS files.

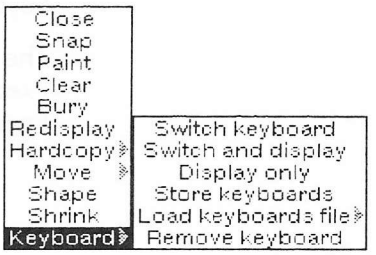
User Interface

Loading VirtualKeyboards adds the item KEYBOARD to the background menu and the default window menu. Using the mouse in this module is the same as in the KeyboardEditor.

Selecting this item from the background menu changes your keyboard for all windows.



Selecting this item from the default window menu allows you to specify a keyboard for an individual window.



The main keyboard module commands are described in detail below.

Switch Keyboards

Use the SWITCH KEYBOARD command to change the behavior of your keyboard to that of a selected virtual keyboard. This brings up a menu of the keyboards currently known to the program.

Select the keyboard you want to substitute for your workstation keyboard. Once you have changed your keyboard's behavior, pressing a key will send the character newly assigned to that key to the current input stream.

Switch & Display a Keyboard

Use the SWITCH AND DISPLAY command to change the behavior of your keyboard to that of a selected keyboard and, in addition, display that keyboard's layout on the screen. You will be offered a menu of the keyboards known to the program; select the one you want to substitute for your workstation's keyboard. Displaying the keyboard layout helps if you're typing on an unfamiliar keyboard. SWITCH AND DISPLAY lets you type characters by using the keyboard displayed on the screen as a menu.

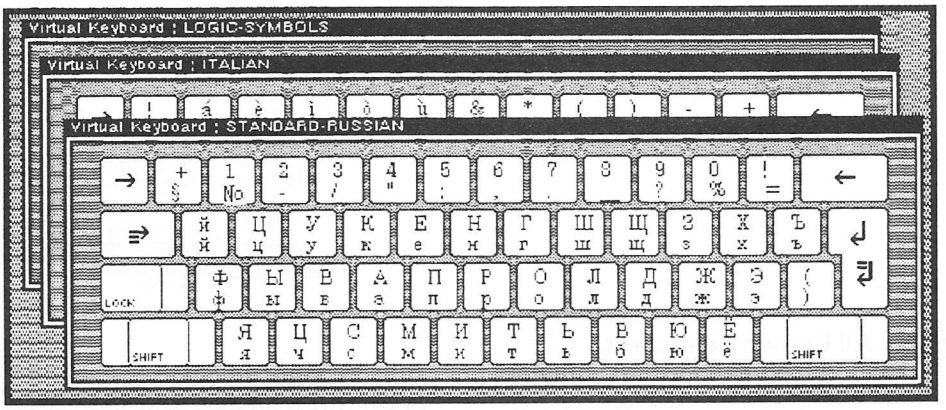


Figure 9. Keyboard layout display

Display-Only a Keyboard

You can display the layout for any given virtual keyboard using the DISPLAY ONLY command. You will be offered a menu of the keyboards known to the program (such as above); select the one you want to display. This is useful if you are primarily using the standard English keyboard but need to type some characters in other languages, or some special characters such as mathematical symbols.

You can use the displayed image as a menu: Selecting a key from the image with the left mouse button will send the character assigned to that key, and pressing the shift key while you click on a key will send the shifted character. Middle-clicking also sends the shifted character.

The effect is exactly as if you had pressed a key with that character assigned to it (except that interrupt characters are treated as ordinary characters; i.e., they do not cause an interrupt). The character is sent to the process that has the TTY (usually where the caret is flashing).

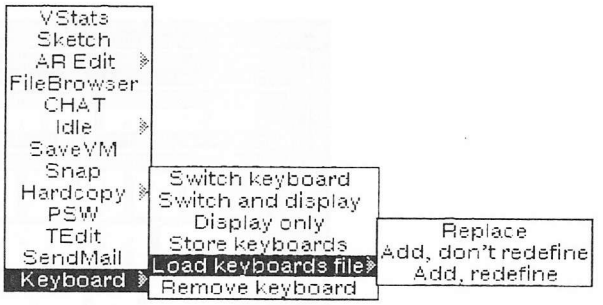
Store Keyboards

After you edit a keyboard (using the KeyboardEditor module), you can store it using the STORE KEYBOARDS command in the top-level menu. When you select STORE KEYBOARDS, the system

will prompt you for a file name. After you type the file name into the prompt window, the system will store all the keyboards known to it (both new and old) in that file in a form that will enable it to load them.

Loading Keyboards File

To load a keyboards file, choose the LOAD KEYBOARDS FILE command, then slide the mouse cursor to the right and choose one of the three items in its submenu.



Replace All Known Keyboards

Choosing REPLACE will load a set of keyboards that you stored using the StORE KEYBOARDS command, replacing all the keyboards currently known to the system.

The currently known keyboards will be lost.

Add New Keyboards to the List of Known Keyboards

To add new keyboards without replacing any of the currently known keyboards, select ADD--DON'T REDEFINE. This will load a set of keyboard definitions.

If a keyboard in the file has the same name as one that is already known to the system, that keyboard will not be loaded and the current definition will stay in effect.

Load New Keyboards and Redefine Existing Keyboards

The ADD--REDEFINE command is similar to ADD--DON'T REDEFINE, except that it redefines existing keyboards that have the same name as keyboards on the file.

Currently known keyboards that do not have the same name as newly loaded keyboards will remain in the list of known keyboards.

Removing Keyboards From the Menu

To remove a keyboard from the set of currently known keyboards, select the REMOVE KEYBOARD command. This will pop up a menu of the known keyboards (such as above), from which you can select a keyboard to be deleted.

Defining a Virtual Keyboard

A virtual keyboard is a list whose CAR is the name of the keyboard and whose CDR is a list of key actions. Creating a new virtual keyboard can be done directly in Lisp or interactively, using the KeyboardEditor module.

Using the Functional Interface

The list of keyboards that are known to the program appears in the menu of keyboard names that pops up when you select SWITCH KEYBOARD from the background menu. This list is stored in the global variable VKBD.KNOWN-KEYBOARDS (see below). To add a keyboard to the list, you have to define that keyboard. To define a keyboard you can either call the function DEFINEKEYBOARD or manipulate the variable VKBD.KNOWN-KEYBOARDS directly as explained herein.

You may also use the KeyboardEditor module, which provides a menu-based user interface for creating and changing keyboard layouts.

A virtual keyboard is a list of the form

(KEYBOARD-NAME *KEY-ASSIGNMENT1 KEY-ASSIGNMENT2* . . .).

A *KEY-ASSIGNMENT* is a list of the form

(*KEY* (*UNSHIFTED-CHAR SHIFTED-CHAR LOCK/UNLOCK*)).

KEY is a key name (the character that appears on the actual keyboard).

UNSHIFTED-CHAR and *SHIFTED-CHAR* are character codes. Each can be either an integer representing the actual code or a list of two elements: the number of the character set and the number of the character in the set.

LOCK/UNLOCK is either the atom LOCKSHIFT, in which case *SHIFTED-CHAR* will be transmitted when the shift-lock key is down, or NOLOCKSHIFT, in which case the shift-lock key has no effect on that key. *LOCK/UNLOCK* is LOCKSHIFT by default.

(DEFINEKEYBOARD *KEYBOARD-NAME LIST-OF-KEY-ASSIGNMENTS KEYS-ARE-NUMBERS?*)

[Function]

Creates a new virtual keyboard after parsing the list of key assignments and adds the keyboard to the list of known keyboards.

If *KEYS-ARE-NUMBERS?* is T, the function expects to find key numbers instead of key names.

(SWITCHKEYBOARDS *NEW-KEYBOARD SWITCH-FLG DISPLAY-FLG MENU-POSITION*)

[Function]

Switches the current keyboard to *NEW-KEYBOARD*, where *NEW-KEYBOARD* is either a virtual keyboard or the name of a known keyboard.

If *SWITCH-FLG* is non-NIL, the actual key actions of the keyboard will be modified.

If *DISPLAY-FLG* is non-NIL, a window with a menu will be displayed. This displayed keyboard will act as a menu and will send characters to the current input stream when a character is selected.

VKBD.KNOWN-KEYBOARDS [Variable]

Contains the list of all currently known virtual keyboards.

Limitations

After loading the Dvorak keyboard, and then restoring defaults, you lose the shift- lock key.

This is a new implementation of a facility similar to but not compatible with the Lyric library module Wherels. Where-Is indexes all definers, but Wherels only indexed Interlisp FNS definitions.

Requirements

Hash-File and Cash-File.

Installation

Load WHERE-IS.DFASL and the required .DFASL modules from the library.

Changed File Manager Functions

Where-Is allows the file manager to know of many more definitions than are actually in the files which have been noticed. In order to achieve this behavior, the following file manager functions are changed when Where-Is is loaded.

Both of these functions are called by the edit interface (the function cl:ed). Thus when Where-Is is loaded the contents of its databases are known to the editor.

(`il:whereis` *name type files filter*) [Function]

Performs as described in the Interlisp Reference Manual. Returns the subset of *files* that contain a *type* definition for *name*. *Files* defaults to il:filelst (all noticed files). When Where-Is is loaded and il:whereis is passed t as its files argument il:whereis will look in the Where-Is databases.

(`il:typesof` *name possible-types impossible-types source filter*) [Function]

Performs as described in the Interlisp Reference Manual. Returns the subset of *possible-types* that *name* is defined as. *Possible-types* defaults to il:filepkgtypes (all define types). When Where-Is is loaded il:typesof will also include the types for *name* in its databases.

Databases

Where-Is provides functions to use and build databases.

Using a Database

(xcl::add-where-is-database *pathname*) [Function]

Adds the database in the file named by *pathname* to the databases known to Where-Is. If a database on an older version of this file is already known, then Where-Is will start using the new version.

(xcl::del-where-is-database *pathname*) [Function]

Deletes the database named by *pathname* from the databases known to Where-Is.

xcl::*where-is-cash-files* [Variable]

Contains the list of databases known to Where-Is.

There is a proceed case for errors while accessing a database which will delete the offending database. This can be useful when a file server goes down.

Building a Database

(xcl::where-is-notice *database-file* &key *files new hash-file-size temp-file-name define-types quiet*) [Function]

Records the definers on *files* in the file named by *database-file*.

Files can be a pathname or a list of pathnames. The default for *files* is "*.;". Note that it is important to include the trailing semi-colon so that only definers on the most recent version are recorded.

If *new* is true a new database file will be created, otherwise *database-file* is presumed to name an existing Where-Is database to be augmented. The default for *new* is nil.

Hash-file-size is only used when new is false and is passed as the *size* argument to make-hash-file. The default for *hash-file-size* is xcl::*where-is-hash-file-size*, which has a default top-level value of 10,000.

If *temp-file-name* is provided then all changes will happen in the temporary file named, which will afterwards be renamed to *database-file*. This can both make things faster (if the temporary file is on a faster device) and doesn't generate a new version of a database until the new version is ready to be used. The use of a temporary file may slow things down when a large existing database is just being updated to reflect a small number of changes.

Define-types is the list of define types (file package types) which should be recorded. The default define types are all those on IL:FILEPKGTYPES which are not aliases for others and which are not in the list xcl::*where-is-ignore-define-types*.

Unless *quiet* is true, xcl::where-is-notice will print the name of each file as it is processed.

xcl::where-is-notice returns the pathname of the hash file written.

[This page intentionally left blank]