Note: This module is provided for backwards compatibility. New applications should use the HASH-FILE Library Module instead of this module.

Hash is a hash-coded dictionary facility, providing much the same functionality as hash arrays do, except that the data is stored in a file.

Hash permits information associated with string or atom keys to be stored on and retrieved from files. The information (or values) associated with the keys in a file may be numbers, strings, or arbitrary Lisp expressions. The associations are maintained by a hashing scheme that minimizes the number of file operations it takes to access a value from its key.

Information is saved in a hash file, which is analogous to a hash array. Actually, hash file can be either the file itself, or the handle on that file which is used by the Hash module. The latter, of data type HashFile, is the datum returned by CREATEHASHFILE or OPENHASHFILE, currently an array record containing the hash file name, the number of slots in the file, the used slots, and other details. All other functions with hash file arguments use this datum.

In older implementations (e.g., for Interlisp-10), hash files came in several varieties, according to the types of value stored in them. The EMYCIN system provided even more flexibility.

This system only supports the most general EXPR type of hash files and EMYCIN-style TEXT entries, in the same file. The VALUETYPE and ITEMLENGTH arguments are for the most part ignored. Two-key hashing is supported in this system but is discouraged as it is only used in EMYCIN, not in the Interlisp-10 system. The functions GETPAGE, DELPAGE, and GETPNAME, which manipulate secret pages, do not exist in this implementation. However, it is permissible to write data at the end of a hash file; that data will be ignored by the Hash module, and can be used to store additional data.

The Hash module views files as a sequence of bytes, randomly accessible. No notice is made of pages, and it is assumed that the host computer buffers I/O sufficiently.

Hash files consist of a short header section (8 bytes), a layer of pointers (4*HASHFILE:Size bytes) followed by ASCII data. Pointers are 3 bytes wide, preceded by a status byte. The pointers point to key PNAMES in the data section, where each key is followed by its value.

Deleted key pointers are reused but deleted data space is not, so rehashing is required if many items have been replaced.

The data section starts at 4*HASHFILE:Size + 9, and consists of alternating keys and values. As deleted data is not rewritten, not all data in the data section is valid.

When a key hashes into a used slot, a probe value is added to it to find the next slot to search. The probe value is a small prime derived from the original hash key.

Requirements

Hash files must reside on a random-access device (not a TCP/IP file server).

Installation

Load HASH.LCOM from the library.

Functions

Creating a Hash File

(CREATEHASHFILE *FILE VALUETYPE ITEMLENGTH #ENTRIES SMASH COPYFN*) [Function]

Creates a new hash file named *FILE*. All other arguments are optional.

VALUETYPE is ignored in this implementation; any hash file can accommodate both Lisp expressions and text.

ITEMLENGTH is not used by the system but is currently saved on the file (if less than 256) for future use.

#ENTRIES is an estimate of the number of entries the file will have. (This should be a realistic guess.)

SMASH is a hash file datum to reuse.

COPYFN is a function to be applied to entries when the file is rehashed (see the description of REHASHFILE, below).

Opening and Closing Hash Files

Before you can use a hash file with this module, you have to open it using the function

(OPENHASHFILE *FILE ACCESS ITEMLENGTH #ENTRIES SMASH*) [Function]

Reopens the previously existing hash file *FILE*.

Access may be INPUT (or NIL), in which case *FILE* is opened for reading only, or BOTH, in which case *FILE* is open for both input and output. Causes an error "not a hashfile", if *FILE* is not recognized as a hash file.

ITEMLENGTH and *#ENTRIES* are for backward compatibility with EMYCIN where OPENHASHFILE also created new hash files; these arguments should be avoided.

SMASH is a hash file datum to reuse.

If *ACCESS* is BOTH and *FILE* is a hash file open for reading only, OPENHASHFILE attempts to close it and reopen it for writing. Otherwise, if *FILE* designates an already open hash file, OPENHASHFILE is a no-op.

OPENHASHFILE returns a hash file datum.

(CLOSEHASHFILE *HASHFILE REOPEN*) [Function]

Closes *HASHFILE* (when you are finished using a hash file, you should close it). If *REOPEN* is non-NIL it should be one of the accepted access types. In this case the file is closed and then immediately reopened with *ACCESS* = *REOPEN*. This is used to make sure the hash file is valid on the disk.

Storing and Retrieving Data

(PUTHASHFILE *KEY VALUE HASHFILE KEY2*) [Function]

Puts *VALUE* under *KEY* in *HASHFILE*. If *VALUE* is NIL, any previous entry for *KEY* is deleted. *KEY2* is for EMYCIN two-key hashing: *KEY2* is internally appended to *KEY* and they are treated as a single key.

(GETHASHFILE *KEY HASHFILE KEY2*) [Function]

Gets the value stored under *KEY IN HASHFILE*. *KEY2* is necessary if it was supplied to PUTHASHFILE.

(LOOKUPHASHFILE *KEY VALUE HASHFILE CALLTYPE KEY2*) [Function]

A generalized entry for inserting and retrieving values; provides certain options not available with GETHASHFILE or PUTHASHFILE. LOOKUPHASHFILE looks up *KEY* in *HASHFILE*.

CALLTYPE is an atom or a list of atoms. The keywords are interpreted as follows:

RETRIEVE — If *KEY* is found, then if *CALLTYPE* is or contains RETRIEVE, the old value is returned from LOOKUPHASHFILE; otherwise returns T.

DELETE — If *CALLTYPE* is or contains DELETE, the value associated with *KEY* is deleted from the file.

REPLACE — If *CALLTYPE* is or contains REPLACE, the old value is replaced with *VALUE*.

INSERT — If *CALLTYPE* is or contains INSERT, LOOKUPHASHFILE inserts *VALUE* as the value associated with *KEY*.

Combinations are possible.

For example, (RETRIEVE DELETE) will delete a key and return the old value.

(PUTHASHTEXT *KEY SRCFIL HASHFILE START END*) [Function]

Puts text from stream *SRCFIL* onto *HASHFILE* under *KEY*. *START* and *END* are passed directly to COPYBYTES.

(GETHASHTEXT *KEY HASHFILE DSTFIL*) [Function]

Uses COPYBYTES to retrieve text stored under *KEY* on *HASHFILE*. The bytes are output to the stream *DSTFIL*.

Functions for Manipulating Hash Files

(HASHFILEP *HASHFILE WRITE?*) [Function]

Returns *HASHFILE* if it is a valid, open hash file datum or returns the hash file datum associated with *HASHFILE* if it is the name of an open hash file. If *WRITE?* is non-NIL, *HASHFILE* must also be open for write access.

(HASHFILEPROP *HASHFILE PROPERTY*) [Function]

Returns the value of a *PROPERTY* of a *HASHFILE* datum. Currently accepted properties are NAME, ACCESS, VALUETYPE, ITEMLENGTH, SIZE, #ENTRIES, COPYFN and STREAM.

(HASHFILENAME *HASHFILE*) [Function]

Same as (HASHFILEPROP *HASHFILE* 'NAME).

(MAPHASHFILE *HASHFILE MAPFN DOUBLE*) [Function]

Maps over *HASHFILE* applying *MAPFN*. If *MAPFN* takes two arguments, it is applied to *KEY* and *VALUE*. If *MAPFN* only takes one argument, it is only applied to *KEY* and saves the cost of reading the value from the file. If *DOUBLE* is non-NIL, then *MAPFN* is applied to *(KEY1 KEY2 VALUE)* or *(KEY1 KEY2)* if the *MAPFN* only takes two arguments.

(REHASHFILE *HASHFILE NEWNAME*) [Function]

As keys are replaced, space in the data section of the file is not reused (though space in the key section is). Eventually the file may need rehashing to reclaim the wasted data space. REHASHFILE is really a special case of COPYHASHFILE, and creates a new file. If *NEWNAME* is non-NIL, it is taken as the name of the rehashed file.

The system automatically rehashes files when 7/8 of the key section is filled. The system will print a message when automatically rehashing a file if the global variable REHASHGAG is non-NIL.

Certain applications save data outside Hash's normal framework. Hash files for those applications will need a custom *COPYFN* (supplied in the call to CREATEHASHFILE), which is used to copy data during the rehashing process. The COPYFN is used as the *FN* argument to COPYHASHFILE during the rehashing.

(COPYHASHFILE *HASHFILE NEWNAME FN VALUETYPE LEAVEOPEN*) [Function]

Makes a copy of *HASHFILE* under *NEWNAME*.

Each key and value pair is moved individually and if *FN* is supplied, is applied to *(KEY VALUE HASHFILE NEWHASHFILE)*.

What it returns is used as the value of the key in the new hash file. (This lets you intervene, perhaps to copy out-of-band data associated with *VALUE*.)

VALUETYPE is a no-op.

If *LEAVEOPEN* is non-NIL then the new hash file datum is returned open, otherwise the new Hash file is closed and the name is returned.

(HASHFILESPLST *HASHFILE XWORD*) [Function]

Returns a Lisp generator for the keys in *HASHFILE*, usable with the spelling corrector. If *XWORD* is supplied, only keys starting with the prefix in *XWORD* are generated.

Global Variables of Hash

HASHFILEDEFAULTSIZE [Variable]

Size used when *#ENTRIES* is omitted or is too small. Default is 512.

HASHFILERDTBL [Variable]

The hash file read table. Default is ORIG.

HASHLOADFACTOR [Variable]

The ratio, used slots/total slots, at which the system rehashes the file. Default is 0.875.

HASHTEXTCHAR [Variable]

The character separating two key hash keys. Default is ↑A.

HFGROWTHFACTOR [Variable]

The ratio of total slots to used slots when a hash file is created. Default is 3.

REHASHGAG [Variable]

Flags whether to print message when rehashing; initially off. Default is NIL.

SYSHASHFILE [Variable]

The current hash file. Default is NIL.

SYSHASHFILELST [Variable]

An alist of open hash files. Default is NIL.

Limitations

The system currently is able to manipulate files on CORE, DSK, FLOPPY and over the network, via leaf servers. Hash files can be used with NS servers only if they support random access files.

Due to the pointer size, only hash files of less than 6 million initial entries can be created, though these can grow to 14 million entries before automatic rehashing exceeds the pointer limit. The total file length is limited to 16 million bytes. No range checking is done for these limits.

Two-key hash files operate on pnames only, without regard to packages.

Hash-File is similar to but not compatible with the library module, Hash. Hash-File is modeled after the Common Lisp hash table facility, and Hash was modeled after the Interlisp hash array facility.

Hash files, like hash tables, are objects which efficiently map from a given Lisp object, called the *key*, to another Lisp object, called the *value*. Hash tables store this mapping in memory, while hash files store the mapping in a specially formatted file. Hash files are generally slower to access than hash tables, but they do not absorb memory and they are persistent over Lisp images. Hash files are recommended for large databases which do not change very often.

Since hash files are not stored in memory, hashing for EQ or EQL keys does not make sense. Memory references written to file in one session will probably not be valid in another. For this reason, the default hashing is for EQUAL keys, and then only those which can be dependably printed and read.

All of the code for Hash-File is in a package called Hash-File. Througout this document Lisp symbols will be printed as though in a package which uses the packages Hash-File and Lisp.

Requirements

Hash files must reside on a random-access device (not a TCP/IP file server).

Installation

Load HASH-FILE.DFASL from the Library.

Functions

Hash-File has functions to create a new hash file, to open and close existing hash files, and to store and retrieve data in hash files.

Creating a Hash File

(make-hash-file *file-name size &key . keys*) [Function]

Creates and returns a new hash file in *file-name* opened for input and output. *Size* indicates the table size and should be an integer somewhat larger than the maximum number of keys under which you expect to store values in this hash file. (The

hash file will grow as required, so this number need not be accurate. See the section , "Rehashing," below.) The keyword arguments are explained as this document progresses.

Opening and Closing Hash Files

(open-hash-file *file-name &key :direction . other-keys*) [Function]

Opens an existing hash file and returns it. The *:direction* argument must be one of `:input` or `:io`. If opened for `:input` then storing values in the hash file will be disallowed. The default for *:direction* is `:input`. Other key arguments are the same as for make-hash-file and are explained as this document progresses.

(close-hash-file *hash-file*) [Function]

Closes the file for *hash-file*, ensuring that all data has been saved. The backing file is always kept coherent; thus the only reason to close the *hash-file* is to ensure that the backing file is properly written to disk. All the functions mentioned in this document which operate on hash files will open the file when necessary; thus it is safe to call `close-hash-file` at almost any time.

Storing and Retrieving Data

(get-hash-file *key hash-file &optional default*) [Function]

Retrieves the value stored under *key* in *hash-file*. Returns *default* if there is nothing stored under *key*. The default for *default* is `nil`. Also returns a second value which is true if something was found under *key* and false otherwise.

(get-hash-file *key hash-file*) [Setf place]

Values can be stores in a hash file with:

(`setf` (`get-hash-file` *key hash-file*) *new-value*)

Accordingly `incf`, `decf`, `push`, `pop` and any other macro that accepts generalized variables will work with `get-hash-file`.

(map-hash-file *function hash-file*) [Function]

For each entry in *hash-file*, *function* is called with the key and value stored.

Note: It is unsafe to change a hash file while mapping over it. The integrity of the file may be lost.

(rem-hash-file *key hash-file*) [Function]

Removes any entry for *key* in *hash-file*. Returns `t` if there was such an entry, `nil` otherwise.

Other Functions

(copy-hash-file *hash-file file-name &optional new-size*) [Function]

Makes and returns a hash file in *file-name* with the same contents as *hash-file*. Much slower than `il:copyfile`, but performs garbage collection, often resulting in a smaller file.

(hash-file-count *hash-file*) [Function]

Returns the number of entries in *hash-file*.

(hash-file-p *object*) [Function]

Returns `t` if *object* is a hash file, `nil` otherwise.

`(hash-file-p` *object*`)` ≡ `(typep` *object* `'hash-file)`

File Format

Hash-File uses a linked bucket implementation as illustrated in Figure 3.

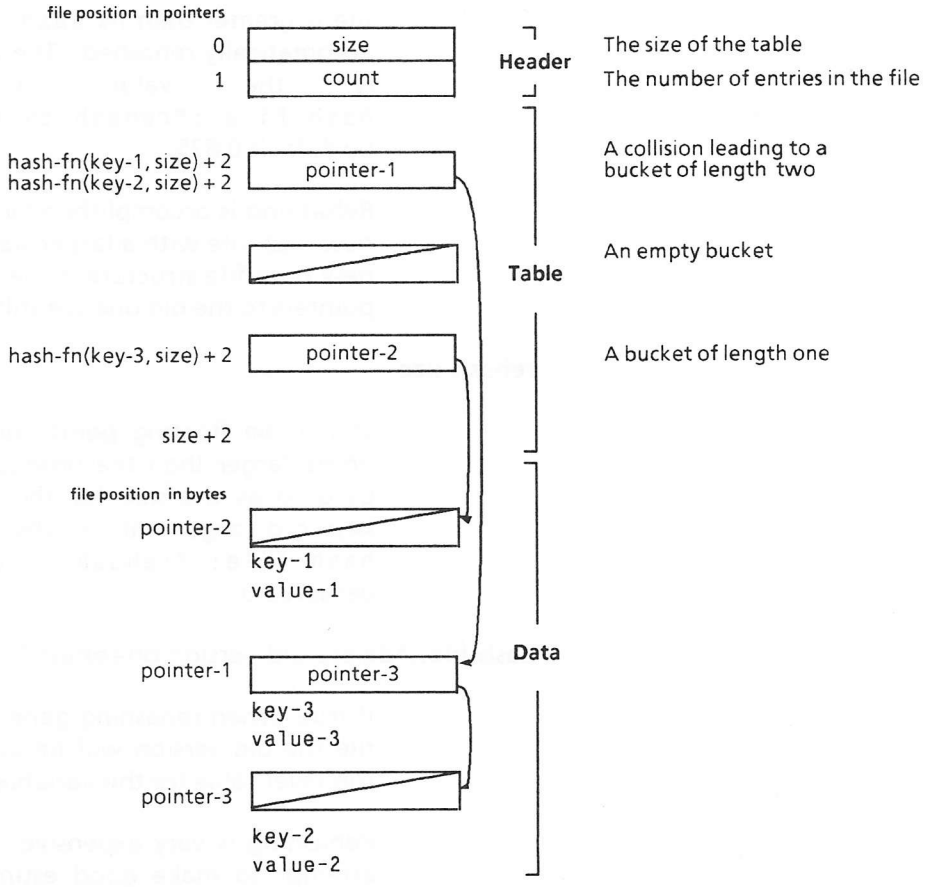


Figure 3. Hash File Format

Pointers are 32-bit integers written as four 8-bit bytes. There are two pointers of header (holding the size and count) followed by *size* pointers of table. Except for in the header and null pointers, all pointers are file-positions in bytes. Every such pointer points to the position on the file of the next pointer in the bucket. Immediately following the next pointer on the file are the printed representation of the key and value for the entry. New entries, including ones for old keys, are always added at the end of the file.

Rehashing

When the number of keys with values in the file reaches a threshold, rehashing is performed to keep bucket lengths from getting too long. This threshold is expressed as a fraction of the table size.

rehash-threshold [Keyword argument]

Should be floating point number between zero and one. When the product of the table size and the rehash threshold of a hash file is greater than its `hash-file-count` then the hash file is automatically rehashed. The default for this keyword argument is the value of the special variable `hash-file::*rehash-threshold*` whose global binding is by default 0.875.

Rehashing is accomplished by having `copy-hash-file` make a new hash file with a larger size in a new version of the file. The new hash file structure is then smashed into the old one so that pointers to the old one are still valid.

rehash-size [Keyword argument]

Should be floating point number larger than one. The next prime larger than the product of this and the old table size is used to as the size for the new table. The default for this keyword argument is the value of the special variable `hash-file::*rehash-size*` whose global binding is by default 2.0.

hash-file::*delete-old-version-on-rehash* [Special variable]

If true, when rehashing generates a new version of the backing file the old version will be automatically deleted. The default top-level value for this variable is `nil`.

Rehashing is very expensive. Thus, when possible, you should attempt to make good estimates for the `size` argument to `make-hash-file`.

Programmer's Interface

There may be applications in which you want to store things in hash files but which could not be printed and read by the functions `print` and `read`. The following hooks are provided for this purpose.

value-read-fn [Keyword argument]

Called by `get-hash-file` with one argument of a stream to read a value. The file position will be set to the same position as it was when this value was written. Default is `hash-file::default-read-fn` which binds `*package*` to the XCL package and `*readtable*` to the XCL readtable before calling `read`.

value-print-fn [Keyword argument]

Called by the setf method for `get-hash-file` with the object to be stored and the stream to print it on. The file position of the stream will be at the end of the file and there are no limitations as to how much can be printed. Default is `hash-file::default-print-fn` which binds `*package*` to the XCL package, `*readtable*` to the XCL readtable and `*print-base*` to 10 before calling `print`.

Example: A hash file with circular values.

```
(defun print-circular-object (object stream)
   (let ((*print-circle* t))
      (hash-file::default-print-fn object stream)))

(setq hash-file-with-circular-values
   (make-hash-file "{core}foo" 10
                                                    :value-print-fn
#'print-circular-object))

(setq l (list "foo"))
(setf (cdr l) l) ⇒ #1= ("foo" . #1#)

(setf (get-hash-file "bar" hash-file-with-circular-values)
l)

(get-hash-file "bar" hash-file-with-circular-values)
   ⇒  #1= ("foo" . #1#)

(eq * l) ⇒ nil
```

key-read-fn [Keyword argument]

Called by `get-hash-file` with one argument of a stream to read a key. The file postion will be set to the same position as it was when this key was written. Default is `hash-file::default-read-fn`, described above.

key-print-fn [Keyword argument]

Called by the setf method for `get-hash-file` with the object to be stored and the stream to print it on. The file position of the stream will be at the end of the file and there are no limitations as to how much can be printed. Default is `hash-file::default-print-fn`, described above.

Note: The value reader is called immediately after the key reader. Thus, the key reader must be sure to read all that the key printer printed so that the file position is appropriate for the value reader. However, the value reader is free to not read all that the value printer printed.

You might now think that you could make a hash file whose keys were circular by simply specifying our circular reader and printer for the key print and read functions, but this would not be sufficient. You also need the following hooks:

key-compare-fn [Keyword argument]

Called when searching a bucket to determine whether the correct key/value pair has been reached yet. Default is `equal`.

key-hash-fn [Keyword argument]

Called with a key and a range. Should return an integer between zero and range-1 with the following property:

key-hash-fn(x) = key-hash-fn(y) iff key-compare-fn(x,y)

The default `key-hash-fn` is `hash-file::hash-object` which works on symbols, strings, lists, bit-vectors, pathnames, characters and numbers. (Any object whose printed representation can be dependably read in as an object equal to the original.)

Note: This function will work on circular lists, as it only proceeds a fixed depth down a structure. Thus to hash on circular keys you also need to provide a key comparer which is able to compare circular keys, as most defintions of equal are not.

Performance

A linked bucket implementation generally gives shorter bucket lengths, but uses more file space. The effects of this upon performance are difficult to judge.

The following table shows the distribution of bucket lengths in a Where-Is hash file containing 27,157 entries with a table size of 50,021.

length	number of buckets this length
0	29,279 (empty buckets)
1	15,461
2	4334
3	794
4	125
5	23
6	4
7	1

This information was gathered by the function `hash-file::histogram`.

[This page intentionally left blank]

HRule is a module that lets you create horizontal rules (solid horizontal lines of various thicknesses) in a TEdit document. Rules are often used to set off titles and page headings from regular text, and to create decorative effects.

Requirements

IMAGEOBJ
EDITBITMAP
TEDIT

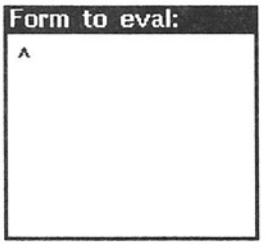
Installation

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

Creating Horizontal Rules

You specify a rule's thickness in decimal fractions of a printer's point (1/72 of an inch).

To create a horizontal rule, place the caret at the point in your document where you want the rule to begin, then type control-O. This will bring up a small window titled "Form to Eval" that contains a blinking caret. Type (HRULE.CREATE *N*) after the caret, with *N* indicating the thickness of the rule.



For example, to create a 4-point rule you would type (HRULE.CREATE 4); to create a $2\frac{1}{2}$-point rule you would type (HRULE.CREATE 2.5). Then press the carriage return. The window will close, and a rule of the specified size will be created, extending from the TEdit caret to the right margin of the paragraph.

Note: This means that nothing can appear to the right of a rule on the same line.

So, for example if you type the following paragraph

This is an example of a paragraph that is about to have a horizontal rule inserted in it, to show what happens.

and insert a $2\frac{1}{2}$-point rule after the word "rule," you end up with

This is an example of a paragraph that is about to have a horizontal rule
inserted in it, to show what happens.

Like other image objects in TEdit, a rule is a single character that can be deleted, moved, and copied like any other character.

You can use the TEdit Paragraph Looks menu to change the width of a rule if you don't want it to extend to the normal right margin of your document.

Stacking Several Rules in a Single Object

Sometimes, you will want to stack several rules atop one another, with space between them. This can be used to achieve effects like

and

To create built-up rules of this type, follow the same procedure as above, but provide a list of rule widths and spacings in place of the single rule width. The first example above was created using the form (HRULE.CREATE '(.5 .5 .5)), and the second example was created using the form (HRULE.CREATE '(3 1 1 1 3)). The first number in the list is the thickness of the topmost rule, the next number is the space below it, the third number is the next rule, and so on.

Limitations

A rule can be, theoretically, infinitely small or infinitely large. For most documents, however, you will probably want to create rules that are between half a point and six points thick. On printers, you can't usually tell the difference between rules that are less than $\frac{1}{2}$ point apart in thickness.

Examples

Shown in Figure 4 are some examples of horizontal rules. In addition, you might want to look at the rules in this document, which were all created with HRule.

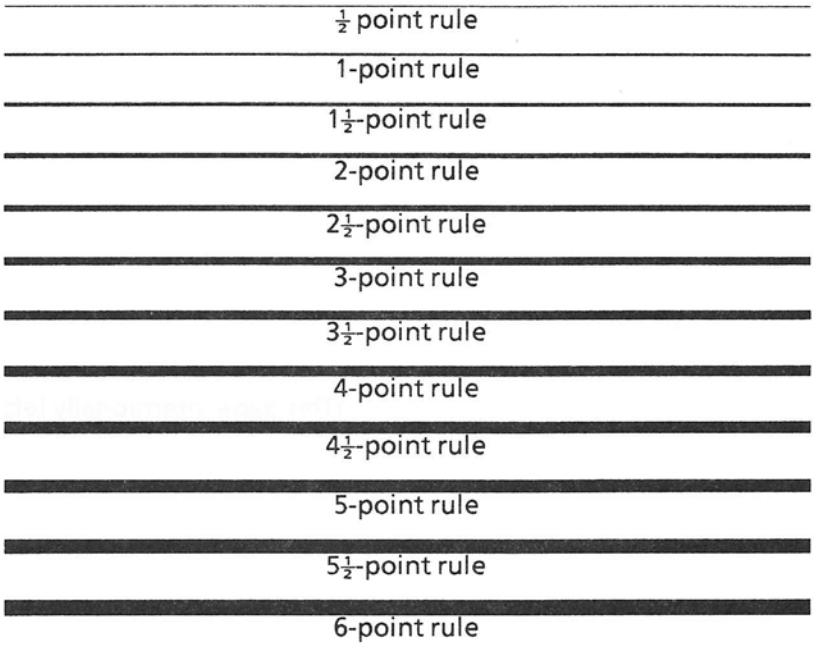


Figure 4. Horizontal rules

Shown in Figure 5 are some examples of built-up rules, along with what you would type to create them:

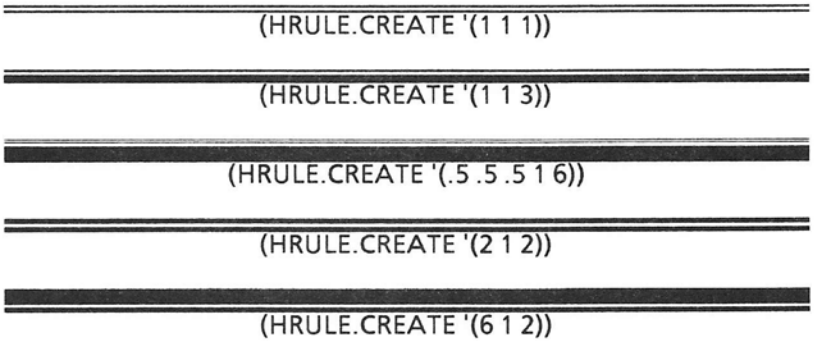


Figure 5. Built-up rules

[This page intentionally left blank]

KERMIT AND MODEM

Kermit and Modem are utilities for transferring files between computers using ordinary RS232 and modem connections.

The file KERMIT.LCOM contains both the Kermit and Modem protocols. Once loaded, it provides a means of transferring files between a Xerox workstation and any other computer that supports either Kermit or Modem, and to which Lisp is able to open a Chat connection.

Of these two file transfer protocols, Kermit is preferred. Modem is much less flexible than Kermit, and cannot be used on RS232 connections requiring parity or flow control. Modem was developed primarily to support file transfers to and from microcomputers running the CP/M operating system. Modem implementations are available for Tops-20, VAX/Unix, and VAX/VMS. Kermit, on the other hand, was designed for file transfers between computers of many types, and there exist implementations of the Kermit protocol on machines ranging in size from eight-bit microcomputers to large IBM mainframes.

For a detailed discussion and tutorial on Kermit, see *Kermit: A File Transfer Protocol* by Frank Da Cruz, Digital Press, 1987.

Requirements

The machine must run Kermit or Modem, and you need the means of reaching it, typically via Chat over an RS232 or a network connection.

You also need the following .LCOM files in order to run this module successfully:

KERMIT, KERMITMENU

as well as CHAT,

and either the RS232C or TCP-IP protocols, or the built-in NS or PUP protocols.

Installation

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

Establishing a Connection

The first step in using Kermit or Modem is to establish a Chat connection with a desired host. You may use any sort of Chat

connection (e.g., NS, TCP, PUP, or RS232). See the Chat module in this manual.

If you are using an RS232 connection, and plan to transfer files with the Modem protocol, do not establish a connection that requires parity to be used; establish the connection with eight bits per character and no parity (see the RS232 module in this manual). Disable flow control (XOn/XOff) when using Modem.

When you have established a Chat connection to a remote host, log in (if necessary) and start the remote host's Kermit or Modem program. The details of running these programs differ slightly between implementations; you should obtain documentation specific to the version of Kermit or Modem running on the remote host.

Kermit

Remote Kermit in Server Mode

Most mainframe implementations of Kermit have a server mode. This mode causes the remote Kermit to listen for either send or receive requests without your having to type additional commands to the remote Kermit. If the version of Kermit you are using on the remote host does support server mode, give the server mode command to place the program in this mode. In most implementations of Kermit, server mode is entered by your typing SERVER to the Kermit prompt:

Kermit>SERVER

Remote Kermit Not in Server Mode

If the remote Kermit does not support server mode, you must issue individual send and receive requests for each file you transfer. To send a file to a remote Kermit, issue the RECEIVE command to the remote Kermit. To receive a file from a remote Kermit, issue the SEND command to the remote Kermit. In most cases, these commands are followed by the name of the file to be sent or received.

For example:

Kermit> RECEIVE *FILENAME*

or

Kermit> SEND *FILENAME*

If you are transferring files between two Xerox workstations connected by an RS232 connection, call (CHAT 'RS232) on each machine to establish the connection. Currently, Lisp Kermit does not support a server mode, so you must issue a receive request on one machine, followed by a send request on the other (see below).

After you have started the remote Kermit program, you need to start the local Lisp Kermit program. Lisp provides both functional and interactive interfaces for Kermit (and Modem).

Local Kermit

To start the local side of the Kermit file transfer, use the KERMIT.SEND or KERMIT.RECEIVE functions:

(KERMIT.SEND *LOCALFILE REMOTEFILE WINDOW TYPE*) [Function]

LOCALFILE is the name of the file being sent to the remote Kermit.

REMOTEFILE is the name under which the file should be stored remotely. In most implementations of Kermit, this name overrides any name you specified in the remote receive command.

WINDOW is a pointer to the Chat window over which the transfer will take place. If *WINDOW* is NIL, the value of CHATWINDOW (the first Chat window to be opened) will be used in its place.

TYPE is the type of the file. It should be set to either TEXT or BINARY.

(KERMIT.RECEIVE *REMOTEFILE LOCALFILE WINDOW TYPE*) [Function]

LOCALFILE is the local name of the file to be received from the remote Kermit.

REMOTEFILE is the name of the file on the remote machine.

WINDOW is a pointer to the Chat window over which the transfer will take place. If *WINDOW* is NIL, the value of CHATWINDOW (the first Chat window to be opened) will be used in its place.

TYPE is the type of the file. It should be set to either TEXT or BINARY.

While the file transfer is in progress, the associated Chat window will be blank, and cumulative packet counts and other messages will be displayed in a one-line prompt window above the Chat window.

Modem

To transfer files with the Modem protocol, you must run the Modem program on the remote machine. Modem does not support a server mode. Typically, you run the program once per file transferred, with instructions in the command line to indicate whether the file is being sent or received. There are a number of versions of the Modem protocol. On some systems, you run the program called Modem; on other systems, the program is called UModem or XModem.

On Unix, for instance, to send a text file to a Xerox workstation, you would type:

%XMODEM -ST *FILENAME*

On Tops-20, you would type:

@MODEM SA *FILENAME*

Note: % and @ are host system prompts.

As with Kermit, after you have started the remote side of the file transfer, you must start the local (Lisp) side. To do this, use either of the functions MODEM.SEND or MODEM.RECEIVE:

(MODEM.SEND *LOCALFILE WINDOW TYPE EOLCONVENTION*) [Function]

LOCALFILE is the name of the file to send to the remote Modem program.

WINDOW is the Chat window over which the transfer will take place.

TYPE is the file type, either TEXT or BINARY.

EOLCONVENTION is the end-of-line convention used by the operating system on which the remote Modem program is running. *EOLCONVENTION* should be one of CR, LF, or CRLF. Typically, Unix and VMS require LF, Tops-20 requires CRLF, and other Xerox machines require CR.

(MODEM.RECEIVE *LOCALFILE WINDOW TYPE EOLCONVENTION*) [Function]

LOCALFILE is the name of the file to receive from the remote Modem program.

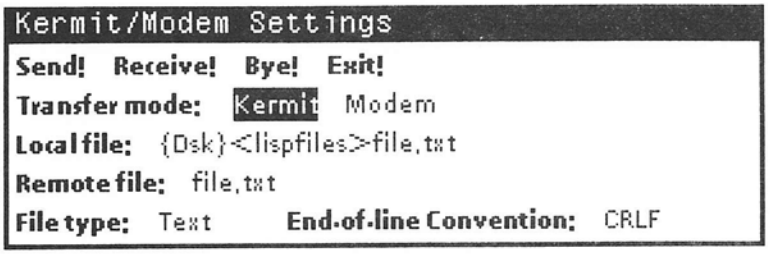
WINDOW is the Chat window over which the transfer will take place.

TYPE is the file type, either TEXT or BINARY.

EOLCONVENTION is the end-of-line convention used by the operating system on which the remote Modem program is running (see above).

Interactive File Transfers With Kermit or Modem

A more convenient user interface for Kermit and Modem is available via the module KERMITMENU.LCOM. It provides a menu-oriented interface for issuing Kermit or Modem commands. To obtain the menu interface, press the middle mouse button in a live Chat window. The standard middle-button Chat menu will contain an entry labeled "Kermit" near its top. If you select this entry, a Kermit menu will appear at the top of the associated Chat window:



The entries on the top line of the menu are action commands:

SEND — Starts sending a file to the remote Kermit or Modem program. The remote program must be prepared to receive the file.

RECEIVE — Starts receiving a file from the remote Kermit or Modem program. The remote program must already be attempting to send the file.

BYE — Closes (severs) the connection.

EXIT — Closes the window containing the menu, but does not close the connection.

TRANSFER MODE — This entry controls whether files are transferred using Kermit or Modem. You may set the state of this entry by selecting either of the Kermit or Modem labels with the mouse. The current transfer mode choice is displayed inverted in the menu.

LOCAL FILE — This entry holds the name of the local file being sent or received. You may set the contents of this field by selecting the LOCAL FILE label and typing the name.

REMOTE FILE — This entry holds the name of the remote file being stored or retrieved. You may set the contents of this field by selecting the REMOTE FILE label and typing the name. The Modem protocol does not use the contents of this field.

FILE TYPE — This field controls whether files are sent in binary or text (ASCII) mode. To set this field, select the FILE TYPE label and choose an entry from the menu that appears.

END-OF-LINE CONVENTION — This field sets the end-of-line convention being used by the remote Modem program (it is not used when files are transferred in Binary mode or with the Kermit protocol). The contents of this field must match the conventions of the operating system on which the remote Modem program is running. To set this field, select the END-OF-LINE CONVENTION label, and choose an entry from the menu that appears.

Limitations

Transfer files between two Xerox machines using the Kermit protocol.

Modem cannot be used on RS232 connections requiring parity or flow control.

[This page intentionally left blank]

KeyboardEditor is intended for use with the VirtualKeyboards module. You should read that module's documentation before reading this. The KeyboardEditor module lets you create new virtual keyboards and change existing ones to suit your needs.

Requirements

VIRTUALKEYBOARDS

Installation

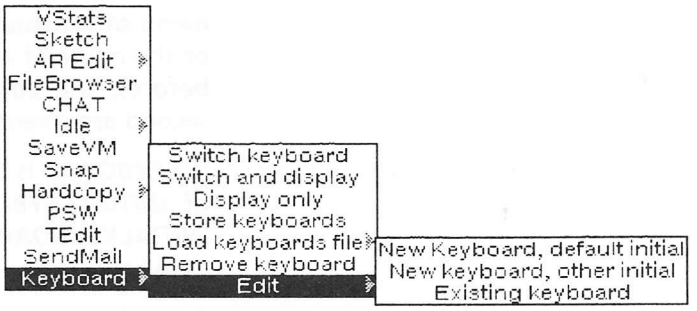
Load KEYBOARDEDITOR.LCOM and VIRTUALKEYBOARDS.LCOM from the library.

User Interface

Loading KeyboardEditor adds EDIT to the Virtual Keyboard submenu on the background menu.

Background Menu

The keyboard editor is used to modify and create virtual keyboards. You can call it by selecting EDIT from the main KeyboardEditor/VirtualKeyboards menu and sliding the cursor to the right to bring up the editor menu. You can also simply select EDIT, which gives you the same options as NEW KEYBOARD, DEFAULT INITIAL.



Creating a New Keyboard From a Copy of the Default Keyboard

Choose NEW KEYBOARD, DEFAULT INITIAL to create a keyboard from a copy of the default keyboard (which initially has the same key assignments as the 1108 keyboard). The system will prompt you for a name for the new keyboard, then call the editor with a copy of the default keyboard as the initial keyboard. The key

assignments that are not changed during the editing session will remain as they are in the default keyboard.

Creating a New Keyboard From a Copy of Any Known Keyboard

To create a new keyboard from a copy of a known keyboard other than the default keyboard, select NEW KEYBOARD, OTHER INITIAL from the Edit submenu. You will be prompted for a name for the new keyboard. The system will then display a menu of the known keyboards to enable you to choose one of them as the initial keyboard.

Changing an Existing Keyboard

You can change an existing keyboard by selecting EXISTING KEYBOARD from the Edit submenu. Like the NEW KEYBOARD, OTHER INITIAL command, this brings up a menu of known keyboards from which you can choose a keyboard for editing. However, you will not be prompted for a keyboard name first, because you are editing the actual keyboard rather than using it as a base for a new keyboard.

Calling the Keyboard Editor From Lisp

The editor can also be called using the function

(EDITKEYBOARD *KEYBOARD INITIALKEYBOARD*) [Function]

where *KEYBOARD* is either a virtual keyboard (i.e., a list) or the name of a virtual keyboard. If *KEYBOARD* is a virtual keyboard or the name of a known keyboard (a keyboard that was defined before), the editing will be done on that keyboard and the second argument will be ignored.

If *KEYBOARD* is a new name, the editing will be done on a copy of *INITIALKEYBOARD*, with *KEYBOARD* as its new name. If *INITIALKEYBOARD* is NIL, the default keyboard will be used as a base keyboard.

Examples:

To create a totally new virtual keyboard, call (EDITKEYBOARD *NEWNAME*).

To create a new keyboard that is similar to a keyboard with the name K1, call (EDITKEYBOARD *NEWNAME* 'K1)

To modify a keyboard with the name GREEK, call (EDITKEYBOARD 'GREEK).

Using the Keyboard Editor

There are four different keyboard editor menus, three of them displayed at any given time. After you call the editor, you will see the command menu at the top, the character menu in the middle, and the keys menu at the bottom.

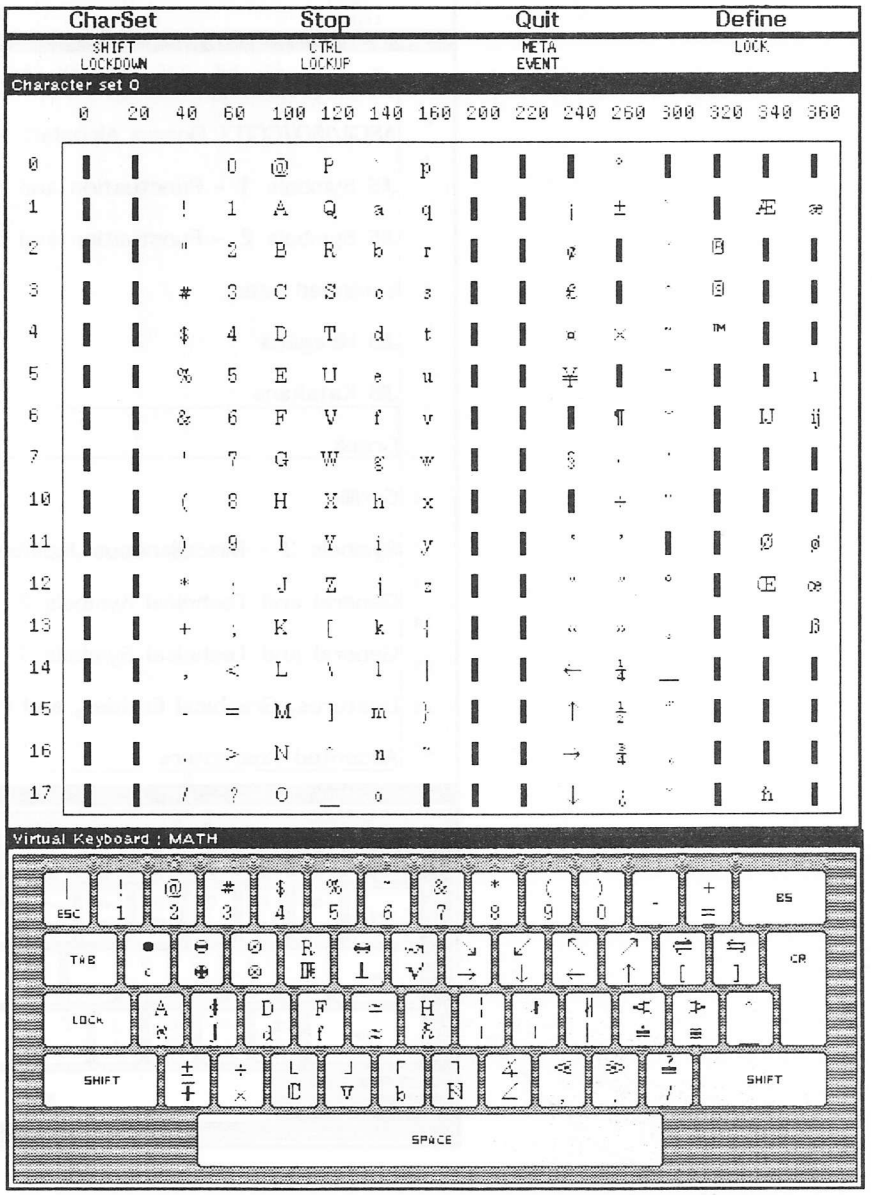


Figure 6. Character Display

The character menu is a 16-by-16-character display of the 256 characters available in the current character set. The set that is displayed when you enter the editor is character set 0, which includes all of the ASCII characters plus many other symbols. See Figure 6. If you need characters from other character sets, you have to select Char Set from the command menu. A new menu will pop up that contains numbers from 0 to 377 octal. This is the character set menu, and it lets you switch the character menu to display characters from other sets. Most of the character set numbers are not currently implemented. The most useful ones are shown in Figure 7.

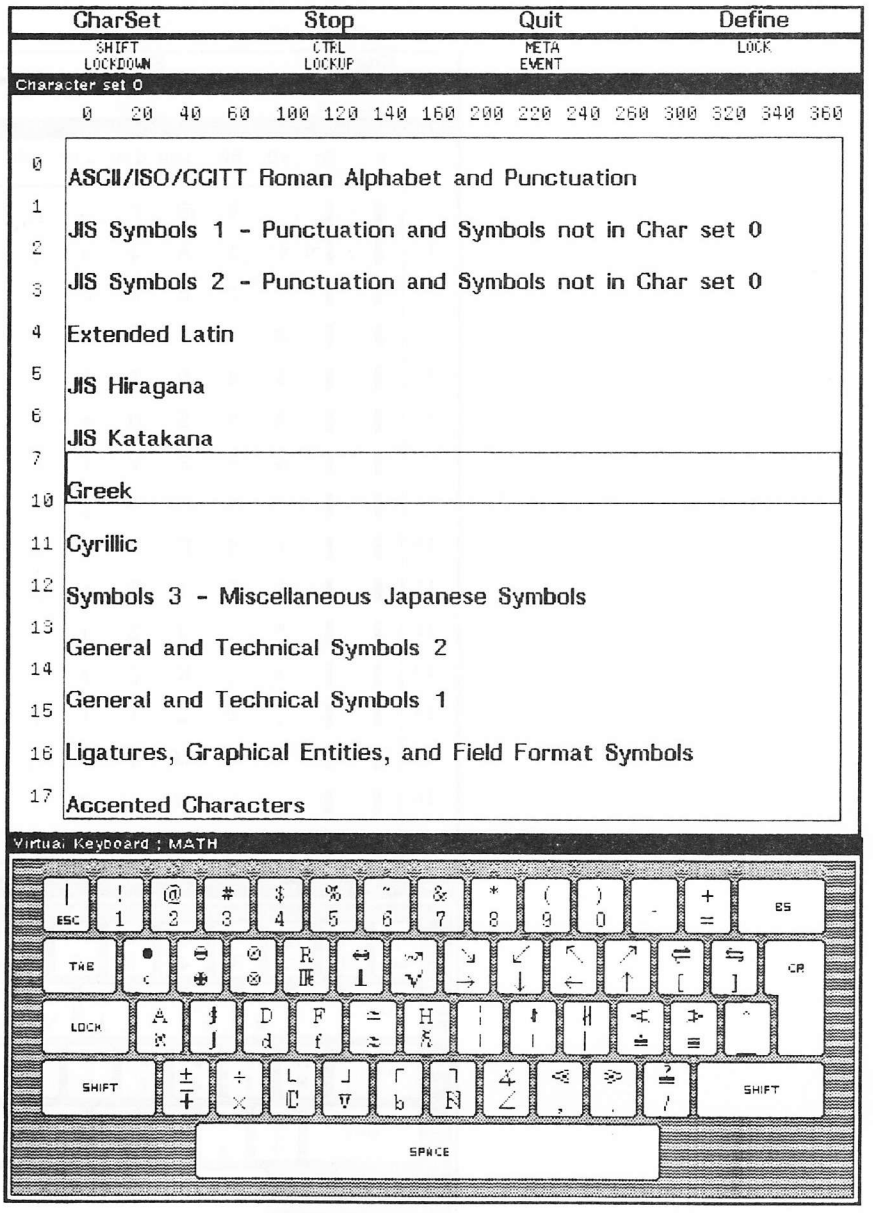


Figure 7. Character Sets

The keys menu lets you make a key the current key by selecting it. A selected key is marked by a black frame. To make a shifted

key the current key, shift-select the key (hold the shift key down and click on the icon with the left button); it will be marked by inverted shift keys in addition to the black frame.

The basic operation of editing is assigning a character to a key. You can only assign character keys; keys other than character keys will retain their current definitions. You assign a character to a key by selecting the key from the keys menu, then selecting the character from the character menu. If the character is to be assigned to the shifted key, select the shifted key as the current key.

A second type of editing operation is to change the LOCKSHIFT state of a key. Each key either has or does not have a LOCKSHIFT property. If a key has a LOCKSHIFT property and the shift lock key of the keyboard is down, typing the key on your workstation keyboard will send the shifted character of the key, regardless of the state of the shift keys. The same rule applies to a virtual displayed keyboard; if the LOCK item is inverted and the key has a LOCKSHIFT property, selecting a key will send the shifted character to the current input stream.

If a key has the LOCKSHIFT property, the lock key will be inverted in the keys menu. To change the LOCKSHIFT property of a key, first make the shifted key the current key. You then set or unset the LOCKSHIFT property by selecting the lock key from the keys menu.

If you are creating a new keyboard and you are satisfied with the key assignments, select Define from the command menu. This will add the newly created keyboard to the list of known keyboards (it will thus appear on future menus). Selecting QUIT will exit after modifying the virtual keyboard, and selecting Stop will exit without modifying the keyboard. In both cases the new keyboard will be returned to the caller of EDITKEYBOARD function (above).

Creating New Keyboard Configurations

KEYBOARDCONFIGURATION [Record]

Describes a physical keyboard: its layout, the key numbers that are used with KEYACTION. It also describes each key: its default meaning, its default label, whether you can change the key's meaning with the keyboard editor.

A configuration consists of a number of parts:

CONFIGURATIONNAME [Record field]

The name of this configuration.

For example, KeyboardEditor comes with configurations named DANDELION (1108), DORADO (1132), DOVE (1186), and FULL-IBMPC.

KEYSIDLIST [Record field]

A list of the IDs you will use for the keys in the rest of the configuration; i.e., your names for the keys. For simplicity, these are usually numbers starting beyond 100 (to avoid overlapping the true range of key numbers).

KEYREGIONS [Record field]

An alist of key IDs and the regions they occupy in the keyboard's image when it is displayed. For example, the alphabetic keys in the DANDELION keyboard are 29 screen points wide and 33 high.

DEFAULTASSIGNMENT [Record field]

An alist of key IDs and their default KEYACTIONs (see *IRM*).

KEYNAMESMAPPING [Record field]

An alist of key names to key IDs. The key names should be mnemonic, and should distinguish relevant differences; e.g., the 7 on the 1186's numeric keypad is named NUMERIC7, while the 7 key in the main keyboard cluster is named 7.

MACHINETYPE [Record field]

The kind of machine for which this configuration is intended.

For example, the FULL-IBMPC configuration is meant to be used with a DAYBREAK keyboard, so its MACHINETYPE is DAYBREAK.

KEYLABELS [Record field]

An alist of key numbers to special labels. This is used to label keys such as the "Next" key, where the key assignment may not be a printable character.

KEYLABELSFONT [Record field]

The font you want to use for the key labels. The default value is Helvetica 5.

BACKGROUNDSHADE [Record field]

The shading for the non-key parts of the virtual keyboard's image. This defaults to a reasonable gray value.

KEYBOARDDISPLAYFONT [Record field]

The font used to display actual character assignments. This should probably be Classic 12, since it is the most complete font.

CHARLABELS [Record field]

An alist from character codes to names. Used to give symbolic names to characters such as ESCAPE, which don't otherwise print.

ACTUALKEYSMAPPING [Record field]

A function that takes one of your key IDs and returns a true key number, for use by KEYACTION.

Note: To create a new configuration, create an instance of the KEYBOARDCONFIGURATION record, using the field names shown above. Then add it to the list

VKBD.CONFIGURATIONS. You may then edit it using the configuration editor described below.

Note: You must save your own configurations. There is no user interface for saving them, nor any automatic scheme.

Editing a Keyboard Configuration

Once you have created a KEYBOARDCONFIGURATION, you can make modest changes to it using the function:

(EDITCONFIGURATION *CONFIGNAME*) [Function]

where *CONFIGNAME* is the CONFIGURATIONNAME you have assigned to your new configuration. This will create a virtual keyboard display window with a menu on top of it as shown in Figure 8.

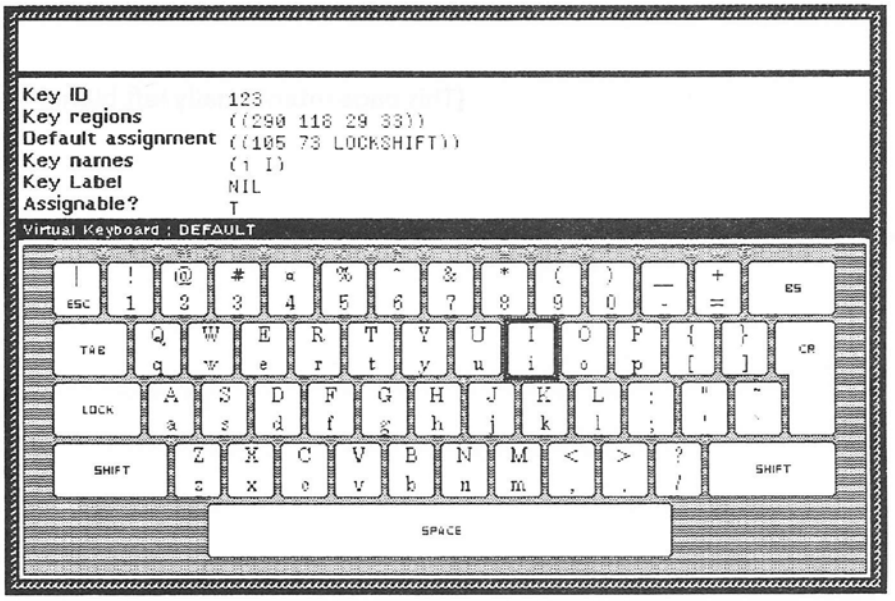


Figure 8. Virtual keyboard display window

Selecting a key with the mouse fills in the fields in the menu. The figure shows the 1108's configuration being edited, with the I key selected. To change one of the values, select the label at the left edge of the menu (e.g., ASSIGNABLE?). You will be prompted to edit the existing value using TTYIN.

The keyboard image is not automatically updated. To refresh it, select REDISPLAY in the right-button window menu.

When you have finished editing, simply close the keyboard window.

[This page intentionally left blank]