HASH-FILE

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 are 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 grows 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 `:in` put then storing values in the hash file is 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 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 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.

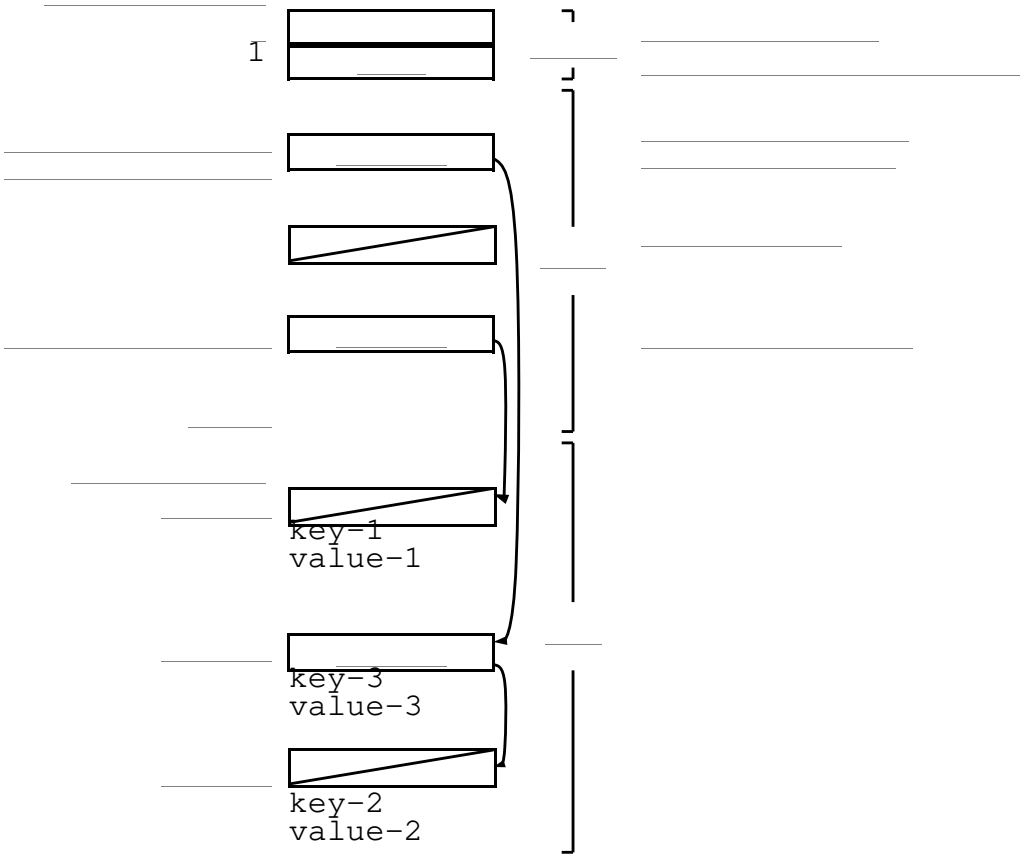

Figure 3. Hash File Format

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 is 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 is 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 is 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 is 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]