

# File: Racket File Format Libraries

Version 5.3

August 6, 2012

## **Contents**

<b>1</b>	<b>Convertible: Data-Conversion Protocol</b>	<b>3</b>
<b>2</b>	<b>gzip Compression and File Creation</b>	<b>5</b>
<b>3</b>	<b>gzip Decompression</b>	<b>6</b>
<b>4</b>	<b>zip File Creation</b>	<b>7</b>
<b>5</b>	<b>tar File Creation</b>	<b>8</b>
<b>6</b>	<b>MD5 Message Digest</b>	<b>9</b>
<b>7</b>	<b>SHA1 Message Digest</b>	<b>10</b>
<b>8</b>	<b>GIF File Writing</b>	<b>11</b>
<b>9</b>	<b>ICO File Reading and Writing</b>	<b>17</b>
<b>10</b>	<b>Windows Registry</b>	<b>19</b>
	<b>Index</b>	<b>22</b>

# 1 Convertible: Data-Conversion Protocol

```
(require file/convertible)
```

The `file/convertible` library provides a protocol to mediate between providers of data in different possible formats and consumers of the formats. For example, a datatype that implements `prop:convertible` might be able to convert itself to a GIF or PDF stream, in which case it would produce data for `'gif-bytes` or `'pdf-bytes` requests.

Any symbol can be used for a conversion request, but the following should be considered standard:

- `'text` — a string for human-readable text
- `'gif-bytes` — a byte string containing a GIF image encoding
- `'png-bytes` — a byte string containing a PNG image encoding
- `'ps-bytes` — a byte string containing a PostScript document
- `'eps-bytes` — a byte string containing an Encapsulated PostScript document
- `'pdf-bytes` — a byte string containing a PDF document
- `'pdf-bytes+bounds` — a list containing a byte string and four numbers; the byte string contains a PDF document and the four numbers are sizing information for the PDF document, namely the width, height, ascent and descent in that order

```
| prop:convertible : struct-type-property?
```

A property whose value should be a procedure of three arguments. The procedure is called when a structure with the property is passed to `convert`; the first argument to the procedure is the structure, the second argument is a symbol for the requested conversion, and the third argument is a value to return (typically `#f` if the conversion is not supported). The procedure's result depends on the requested conversion.

```
| (convertible? v) → boolean?  
  v : any/c
```

Returns `#t` if `v` supports the conversion protocol, `#f` otherwise.

```
| (convert v request [default])
```

```

→ (case request
    [(text) (or/c string? (λ (x) (eq? x default)))]
    [(gif-bytes png-bytes ps-bytes eps-bytes pdf-bytes)
     (or/c bytes? (λ (x) (eq? x default)))]
    [(pdf-bytes+bounds) (or/c (list/c bytes?
                                   (and/c real? (not/c negative?))
                                   (and/c real? (not/c negative?))
                                   (and/c real? (not/c negative?))
                                   (and/c real? (not/c negative?)))
      (λ (x) (eq? x default)))]
    [else any/c])
  v : convertible?
  request : symbol?
  default : any/c = #f

```

Requests a data conversion from *v*, where *request* indicates the type of requested data and *default* is the value that the converter should return if it cannot produce data in the format indicated by *request*.

## 2 gzip Compression and File Creation

```
(require file/gzip)
```

The `file/gzip` library provides utilities to create archive files in gzip format, or simply to compress data using the pkzip “deflate” method.

```
(gzip in-file [out-file]) → void?  
  in-file : path-string?  
  out-file : path-string? = (string-append in-file ".gz")
```

Compresses data to the same format as the `gzip` utility, writing the compressed data directly to a file. The `in-file` argument is the name of the file to compress. If the file named by `out-file` exists, it will be overwritten.

```
(gzip-through-ports in  
                    out  
                    orig-filename  
                    timestamp) → void?  
  in : input-port?  
  out : output-port?  
  orig-filename : (or/c string? false/c)  
  timestamp : exact-integer?
```

Reads the port `in` for data and compresses it to `out`, outputting the same format as the `gzip` utility. The `orig-filename` string is embedded in this output; `orig-filename` can be `#f` to omit the filename from the compressed stream. The `timestamp` number is also embedded in the output stream, as the modification date of the original file (in Unix seconds, as `file-or-directory-modify-seconds` would report on Unix).

```
(deflate in out) → exact-nonnegative-integer?  
                  exact-nonnegative-integer?  
                  exact-nonnegative-integer?  
  in : input-port?  
  out : output-port?
```

Writes pkzip-format “deflated” data to the port `out`, compressing data from the port `in`. The data in a file created by `gzip` uses this format (preceded with header information).

The result is three values: the number of bytes read from `in`, the number of bytes written to `out`, and a cyclic redundancy check (CRC) value for the input.

### 3 gzip Decompression

```
(require file/gunzip)
```

The `file/gunzip` library provides utilities to decompress archive files in gzip format, or simply to decompress data using the pkzip “inflate” method.

```
(gunzip file [output-name-filter]) → void?  
file : path-string?  
output-name-filter : (string? boolean? . -> . path-string?)  
                    = (lambda (file archive-supplied?) file)
```

Extracts data that was compressed using the gzip utility (or `gzip` function), writing the uncompressed data directly to a file. The `file` argument is the name of the file containing compressed data. The default output file name is the original name of the compressed file as stored in `file`. If a file by this name exists, it will be overwritten. If no original name is stored in the source file, “unzipped” is used as the default output file name.

The `output-name-filter` procedure is applied to two arguments—the default destination file name and a boolean that is `#t` if this name was read from `file`—before the destination file is created. The return value of the file is used as the actual destination file name (to be opened with the `'truncate` flag of `open-output-file`).

If the compressed data turns out to be corrupted, the `exn:fail` exception is raised.

```
(gunzip-through-ports in out) → void?  
in : input-port?  
out : output-port?
```

Reads the port `in` for compressed data that was created using the gzip utility, writing the uncompressed data to the port `out`.

If the compressed data turns out to be corrupted, the `exn:fail` exception is raised. The unzipping process may peek further into `in` than needed to decompress the data, but it will not consume the unneeded bytes.

```
(inflate in out) → void?  
in : input-port?  
out : output-port?
```

Reads pkzip-format “deflated” data from the port `in` and writes the uncompressed (“inflated”) data to the port `out`. The data in a file created by gzip uses this format (preceded with some header information).

If the compressed data turns out to be corrupted, the `exn:fail` exception is raised. The inflate process may peek further into `in` than needed to decompress the data, but it will not consume the unneeded bytes.

## 4 zip File Creation

```
(require file/zip)
```

The `file/zip` library provides utilities to create zip archive files, which are compatible with both Windows and Unix (including Mac OS X) unpacking. The actual compression is implemented by `deflate`.

```
(zip zip-file path ...) → void?  
  zip-file : path-string?  
  path : path-string?
```

Creates `zip-file`, which holds the complete content of all `paths`. The given `paths` are all expected to be relative path names of existing directories and files (i.e., relative to the current directory). If a nested path is provided as a `path`, its ancestor directories are also added to the resulting zip file, up to the current directory (using `pathlist-closure`). Files are packaged as usual for zip files, including permission bits for both Windows and Unix (including Mac OS X). The permission bits are determined by `file-or-directory-permissions`, which does not preserve the distinction between owner/group/other permissions. Also, symbolic links are always followed.

```
(zip->output paths [out]) → void?  
  paths : (listof path-string?)  
  out : output-port? = (current-output-port)
```

Zips each of the given `paths`, and packages it as a zip “file” that is written directly to `out`. Unlike `zip`, the specified `paths` are included as-is; if a directory is specified, its content is not automatically added, and nested directories are added without parent directories.

```
(zip-verbose) → boolean?  
(zip-verbose on?) → void?  
  on? : any/c
```

A parameter that controls output during a `zip` operation. Setting this parameter to a true value causes `zip` to display to `(current-error-port)` the filename that is currently being compressed.

## 5 tar File Creation

```
(require file/tar)
```

The `file/tar` library provides utilities to create archive files in USTAR format, like the archive that the Unix utility `pax` generates. The USTAR format imposes limits on path lengths. The resulting archives contain only directories and files (symbolic links are followed), and owner information is not preserved; the owner that is stored in the archive is always “root.”

```
(tar tar-file path ...) → exact-nonnegative-integer?  
tar-file : path-string?  
path : path-string?
```

Creates `tar-file`, which holds the complete content of all `paths`. The given `paths` are all expected to be relative path names of existing directories and files (i.e., relative to the current directory). If a nested path is provided as a `path`, its ancestor directories are also added to the resulting tar file, up to the current directory (using `pathlist-closure`).

```
(tar->output paths [out]) → exact-nonnegative-integer?  
paths : (listof path?)  
out : output-port? = (current-output-port)
```

Packages each of the given `paths` in a tar format archive that is written directly to the `out`. The specified `paths` are included as-is; if a directory is specified, its content is not automatically added, and nested directories are added without parent directories.

```
(tar-gzip tar-file paths ...) → void?  
tar-file : path-string?  
paths : path-string?
```

Like `tar`, but compresses the resulting file with `gzip`.



## 6 MD5 Message Digest

```
(require file/md5)

(md5 in [hex-encode?]) → bytes?
  in : (or/c input-port? bytes? string?)
  hex-encode? : boolean? = #t
```

If `hex-encode?` is `#t`, produces a byte string containing 32 hexadecimal digits (lowercase) that is the MD5 hash of the given input stream or byte string. Otherwise produces the 16 byte long byte string that is the MD5 hash of the given input stream or byte string.

Examples:

```
> (md5 #"abc")
#"900150983cd24fb0d6963f7d28e17f72"
> (md5 #"abc" #f)
#"\\220\\1P\\230<\\3220\\260\\326\\226?}\\341\\177r"
```

## 7 SHA1 Message Digest

```
(require file/sha1)
```

See `openssl/sha1` for a faster implementation.

```
(sha1 in) → string?  
in : input-port
```

Returns a 40-character string that represents the SHA-1 hash (in hexadecimal notation) of the content from `in`, consuming all of the input from `in` until an end-of-file.

The `sha1` function composes `bytes->hex-string` with `sha1-bytes`.

Example:

```
> (sha1 (open-input-bytes #"abc"))  
"a9993e364706816aba3e25717850c26c9cd0d89d"
```

```
(sha1-bytes in) → bytes?  
in : input-port
```

Returns a 20-byte byte string that represents the SHA-1 hash of the content from `in`, consuming all of the input from `in` until an end-of-file.

Example:

```
> (sha1-bytes (open-input-bytes #"abc"))  
#\251\231>6G\6\201j\272>%qXP\302l\234\320\330\235"
```

```
(bytes->hex-string bstr) → string?  
bstr : bytes?
```

Converts the given byte string to a string representation, where each byte in `bstr` is converted to its two-digit hexadecimal representation in the resulting string.

## 8 GIF File Writing

```
(require file/gif)
```

The `file/gif` library provides functions for writing GIF files to a stream, including GIF files with multiple images and controls (such as animated GIFs).

A GIF stream is created by `gif-start`, and then individual images are written with `gif-add-image`. Optionally, `gif-add-control` inserts instructions for rendering the images. The `gif-end` function ends the GIF stream.

A GIF stream can be in any one of the following states:

- `'init` : no images or controls have been added to the stream
- `'image-or-control` : another image or control can be written
- `'image` : another image can be written (but not a control, since a control was written)
- `'done` : nothing more can be added

```
(gif-stream? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a GIF stream created by `gif-write`, `#f` otherwise.

```
(image-ready-gif-stream? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a GIF stream that is not in `'done` mode, `#f` otherwise.

```
(image-or-control-ready-gif-stream? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a GIF stream that is in `'init` or `'image-or-control` mode, `#f` otherwise.

```
(empty-gif-stream? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a GIF stream that in `'init` mode, `#f` otherwise.

```
(gif-colormap? v) → boolean?  
v : any/c
```

Returns `#t` if `v` represents a colormap, `#f` otherwise. A colormap is a list whose size is a power of 2 between  $2^1$  and  $2^8$ , and whose elements are vectors of size 3 containing colors (i.e., exact integers between 0 and 255 inclusive).

```
(color? v) → boolean?  
v : any/c
```

The same as `byte?`.

```
(dimension? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is an exact integer between 0 and 65535 inclusive, `#f` otherwise.

```
(gif-state stream) → symbol?  
stream : gif-stream?
```

Returns the state of `stream`.

```
(gif-start out w h bg-color cmap) → gif-stream?  
out : output-port?  
w : dimension?  
h : dimension?  
bg-color : color?  
cmap : (or/c gif-colormap? #f)
```

Writes the start of a GIF file to the given output port, and returns a GIF stream that adds to the output port.

The width and height determine a virtual space for the overall GIF image. Individual images added to the GIF stream must fit within this virtual space. The space is initialized by the given background color.

Finally, the default meaning of color numbers (such as the background color) is determined by the given colormap, but individual images within the GIF file can have their own colormaps.

A global colormap need not be supplied, in which case a colormap must be supplied for each image. Beware that the `bg-color` is ill-defined if a global colormap is not provided.

```
(gif-add-image stream
  left
  top
  width
  height
  interlaced?
  cmap
  bstr) → void?
stream : image-ready-gif-stream?
left : dimension?
top : dimension?
width : dimension?
height : dimension?
interlaced? : any/c
cmap : (or/c gif-colormap? #f)
bstr : bytes?
```

Writes an image to the given GIF stream. The *left*, *top*, *width*, and *height* values specify the location and size of the image within the overall GIF image's virtual space.

If *interlaced?* is true, then *bstr* should provide bytes in interlaced order instead of top-to-bottom order. Interlaced order is:

- every 8th row, starting with 0
- every 8th row, starting with 4
- every 4th row, starting with 2
- every 2nd row, starting with 1

If a global color is provided with *gif-start*, a *#f* value can be provided for *cmap*.

The *bstr* argument specifies the pixel content of the image. Each byte specifies a color (i.e., an index in the colormap). Each row is provided left-to-right, and the rows provided either top-to-bottom or in interlaced order (see above). If the image is prefixed with a control that specifies an transparent index (see *gif-add-control*), then the corresponding “color” doesn't draw into the overall GIF image.

An exception is raised if any byte value in *bstr* is larger than the colormap's length, if the *bstr* length is not *width* times *height*, or if the *top*, *left*, *width*, and *height* dimensions specify a region beyond the overall GIF image's virtual space.

```
(gif-add-control stream
      disposal
      wait-for-input?
      delay
      transparent) → void?
stream : image-or-control-ready-gif-stream?
disposal : (or/c 'any 'keep 'restore-bg 'restore-prev)
wait-for-input? : any/c
delay : dimension?
transparent : (or/c color? #f)
```

Writes an image-control command to a GIF stream. Such a control must appear just before an image, and it applies to the following image.

The GIF image model involves processing images one by one, placing each image into the specified position within the overall image's virtual space. An image-control command can specify a delay before an image is added (to create animated GIFs), and it also specifies how the image should be kept or removed from the overall image before proceeding to the next one (also for GIF animation).

The *disposal* argument specifies how to proceed:

- *'any* : doesn't matter (perhaps because the next image completely overwrites the current one)
- *'keep* : leave the image in place
- *'restore-bg* : replace the image with the background color
- *'restore-prev* : restore the overall image content to the content before the image is added

If *wait-for-input?* is true, then the display program may wait for some cue from the user (perhaps a mouse click) before adding the image.

The *delay* argument specifies a delay in 1/100s of a second.

If the *transparent* argument is a color, then it determines an index that is used to represent transparent pixels in the follow image (as opposed to the color specified by the colormap for the index).

An exception is raised if a control is already added to *stream* without a corresponding image.

```
(gif-add-loop-control stream iteration) → void?
stream : empty-gif-stream?
iteration : dimension?
```

Writes a control command to a GIF stream for which no images or other commands have already been written. The command causes the animating sequence of images in the GIF to be repeated ‘iteration-dimension’ times, where 0 can be used to mean “infinity.”

An exception is raised if some control or image has been added to the stream already.

```
(gif-add-comment stream bstr) → void?  
  stream : image-or-control-ready-gif-stream?  
  bstr : bytes?
```

Adds a generic comment to the GIF stream.

An exception is raised if an image-control command was just written to the stream (so that an image is required next).

```
(gif-end stream) → void?  
  stream : image-or-control-ready-gif-stream?
```

Finishes writing a GIF file. The GIF stream’s output port is not automatically closed.

An exception is raised if an image-control command was just written to the stream (so that an image is required next).

```
(quantize bstr) → bytes? gif-colormap? (or/c color? #f)  
  (and/c bytes?  
    bstr : (lambda (bstr)  
              (zero? (remainder (bytes-length bstr) 4))))
```

Each image in a GIF stream is limited to 256 colors, including the transparent “color,” if any. The `quantize` function converts a 24-bit image (plus alpha channel) into an indexed-color image, reducing the number of colors if necessary.

Given a set of pixels expressed in ARGB format (i.e., each four bytes is a set of values for one pixel: alpha, red, blue, and green), `quantize` produces

- bytes for the image (i.e., a array of colors, expressed as a byte string)
- a colormap
- either `#f` or a color index for the transparent “color”

The conversion treats alpha values less than 128 as transparent pixels, and other alpha values as solid.

The quantization process uses Octrees [Gervautz1990] to construct an adaptive palette for all (non-transparent) colors in the image. This implementation is based on an article by Dean Clark [Clark1996].

To convert a collection of images all with the same quantization, simply append them for the input of a single call of `quantize`, and then break apart the result bytes.



## 9 ICO File Reading and Writing

```
(require file/ico)
```

The `file/ico` library provides functions for reading and writing ".ico" files, which contain one or more icons. Each icon is up to 256 by 256 pixels, has a particular depth (i.e., bits per pixel used to represent a color), and mask (i.e., whether a pixel is shown, except that the mask may be ignored for 32-bit icons that have an alpha value per pixel).

```
(ico? v) → boolean?  
v : any/c
```

Returns `#t` if `v` represents an icon, `#f` otherwise.

```
(ico-width ico) → (integer-in 1 256)  
ico : ico?  
(ico-height ico) → (integer-in 1 256)  
ico : ico?  
(ico-depth ico) → (one-of/c 1 2 4 8 16 24 32)  
ico : ico?
```

Returns the width or of an icon in pixels, or the depth in bits per pixel.

```
(read-icos src) → (listof ico?)  
src : (or/c path-string? input-port?)
```

Parses `src` as an ".ico" to extract a list of icons.

```
(read-icos-from-exe src) → (listof ico?)  
src : (or/c path-string? input-port?)
```

Parses `src` as an ".exe" to extract the list of icons that represent the Windows executable.

```
(write-icos icos dest [#:exists exists]) → void?  
icos : (listof ico?)  
dest : (or/c path-string? output-port?)  
      (or/c 'error 'append 'update 'can-update  
exists : 'replace 'truncate = 'error  
         'must-truncate 'truncate/replace)
```

Writes each icon in `icos` to `dest` as an ".ico" file. If `dest` is not an output port, `exists` is passed on to `open-output-file` to open `dest` for writing.

```
(replace-icos icos dest) → void?  
icos : (listof ico?)  
dest : (or/c path-string? output-port?)
```

Writes icons in *icos* to replace icons in *dest* as an Windows executable. Only existing icon sizes and depths in the executable are replaced, and best matches for the existing sizes and depth are drawn from *icos* (adjusting the scale and depth if a best match as necessary).

```
(ico->argb ico) → bytes?  
ico : ico?
```

Converts an icon to an ARGB byte string, which has the icon's pixels in left-to-right, top-to-bottom order, with four bytes (alpha, red, green, and blue channels) for each pixel.

```
(argb->ico width height bstr [#:depth depth]) → ico?  
width : (integer-in 1 256)  
height : (integer-in 1 256)  
bstr : bytes?  
depth : (one-of/c 1 2 4 8 24 32) = 32
```

Converts an ARGB byte string (in the same format as from *ico->argb*) to an icon of the given width, height, and depth.

The *bstr* argument must have a length *( \* 4 width height )*, and *( \* width depth )* must be a multiple of 8.

## 10 Windows Registry

```
(require file/resource)

(get-resource section
  entry
  [value-box
   file
   #:type type])
→ (or/c #f string? bytes? exact-integer? #t)
section : string?
entry : string?
value-box : (or/f #f (box/c (or/c string? bytes? exact-integer?)))
           = #f
file : (or/c #f fail-path?) = #f
type : (or/c 'string 'bytes 'integer) = derived-from-value-box
```

Gets a value from the Windows registry or an ".ini" file. For backward compatibility, the result is `#f` for platforms other than Windows. The registry is read when `file` is `#f` and when `section` is "HKEY\_CLASSES\_ROOT", "HKEY\_CURRENT\_CONFIG", "HKEY\_CURRENT\_USER", "HKEY\_LOCAL\_MACHINE", or "HKEY\_USERS". When `file` is `#f` and `section` is not one of the special registry strings, then `(build-path (find-system-path 'home-dir) "mred.ini")` is read.

The resource value is keyed on the combination of `section` and `entry`. The result is `#f` if no value is found for the specified `section` and `entry`. If `value-box` is a box, then the result is `#t` if a value is found, and the box is filled with the value; when `value-box` is `#f`, the result is the found value.

The `type` argument determines how a value in the resource is converted to a Racket value. If `value-box` is a box, then the default `type` is derived from the initial box content, otherwise the default `type` is `'string`.

Registry values of any format can be extracted. Values using the registry format REG\_SZ are treated as strings, and values with the format REG\_DWORD are treated as 32-bit signed integers. All other formats are treated as raw bytes. Data from the registry is converted to the requested `type` as follows:

- A REG\_SZ registry value is converted to an integer using `string->number` (using 0 if the result is not an exact integer), and it is converted to bytes using `string->bytes/utf-8`.
- A REG\_DWORD registry value is converted to a string or byte string via `number->string` and (for byte strings) `string->bytes/utf-8`.
- Any other kind of registry value is converted to a string or integer using `bytes->string/utf-8` and (for integers) `string->number`.

Resources from ".ini" files are always strings, and are converted like REG\_SZ registry values.

To get the “default” value for a registry entry, use a trailing backslash. For example, the following expression gets a command line for starting a browser:

```
(get-resource "HKEY_CLASSES_ROOT"
              "htmlfile\\shell\\open\\command\\")
```

```
(write-resource section
                entry
                value
                [file
                 #:type type
                 #:create-key? create-key?]) → boolean?

section : string?
entry : string?
value : (or/c string? bytes? exact-integer?)
file : (or/c path-string? #f) = #f
type : (or/c 'string 'bytes 'integer) = 'string
create-key? : any/c = #f
```

Write a value to the Windows registry or an ".ini" file. For backward compatibility, the result is `#f` for platforms other than Windows. The registry is written when `file` is `#f` and when `section` is "HKEY\_CLASSES\_ROOT", "HKEY\_CURRENT\_CONFIG", "HKEY\_CURRENT\_USER", "HKEY\_LOCAL\_MACHINE", or "HKEY\_USERS". When `file` is `#f` and `section` is not one of the special registry strings, then `(build-path (find-system-path 'home-dir) "mred.ini")` is written.

The resource value is keyed on the combination of `section` and `entry`. If `create-key?` is false when writing to the registry, the resource entry must already exist, otherwise the write fails. The result is `#f` if the write fails or `#t` if it succeeds.

The `type` argument determines the format of the value written to the registry: `'string` writes using the REG\_SZ format, `'bytes` writes using the REG\_BINARY format, and `'dword` writes using the REG\_DWORD format. Any kind of `value` can be converted for any kind of `type` using the inverse of the conversions for `get-resource`.

When writing to an ".ini" file, the format is always a string, independent of `type`.

## **Bibliography**

- [Gervautz1990] M. Gervautz and W. Purgathofer, "A simple method for color quantization: Octree quantization," Graphics Gems, 1990.
- [Clark1996] Dean Clark, "Color Quantization using Octrees," Dr. Dobbs Journal, January 1, 1996. <http://www.ddj.com/184409805>

## Index

- [argb->ico](#), 18
- [bytes->hex-string](#), 10
- [color?](#), 12
- [convert](#), 3
- Convertible: Data-Conversion Protocol, 3
- [convertible?](#), 3
- [deflate](#), 5
- [dimension?](#), 12
- [empty-gif-stream?](#), 11
- [file/convertible](#), 3
- [file/gif](#), 11
- [file/gunzip](#), 6
- [file/gzip](#), 5
- [file/ico](#), 17
- [file/md5](#), 9
- [file/resource](#), 19
- [file/sha1](#), 10
- [file/tar](#), 8
- [file/zip](#), 7
- File: Racket File Format Libraries, 1
- [get-resource](#), 19
- GIF File Writing, 11
- [gif-add-comment](#), 15
- [gif-add-control](#), 14
- [gif-add-image](#), 13
- [gif-add-loop-control](#), 14
- [gif-colormap?](#), 12
- [gif-end](#), 15
- [gif-start](#), 12
- [gif-state](#), 12
- [gif-stream?](#), 11
- [gunzip](#), 6
- [gunzip-through-ports](#), 6
- [gzip](#), 5
- gzip Compression and File Creation, 5
- gzip Decompression, 6
- [gzip-through-ports](#), 5
- ["HKEY\\_CLASSES\\_ROOT"](#), 19
- ["HKEY\\_CURRENT\\_CONFIG"](#), 19
- ["HKEY\\_CURRENT\\_USER"](#), 19
- ["HKEY\\_LOCAL\\_MACHINE"](#), 19
- ["HKEY\\_USERS"](#), 19
- ICO File Reading and Writing, 17
- [ico->argb](#), 18
- [ico-depth](#), 17
- [ico-height](#), 17
- [ico-width](#), 17
- [ico?](#), 17
- [image-or-control-ready-gif-stream?](#), 11
- [image-ready-gif-stream?](#), 11
- [inflate](#), 6
- [md5](#), 9
- MD5 Message Digest, 9
- [prop:convertible](#), 3
- [quantize](#), 15
- [read-icos](#), 17
- [read-icos-from-exe](#), 17
- [replace-icos](#), 17
- [sha1](#), 10
- SHA1 Message Digest, 10
- [sha1-bytes](#), 10
- [tar](#), 8
- tar File Creation, 8
- [tar->output](#), 8
- [tar-gzip](#), 8
- Windows Registry, 19
- [write-icos](#), 17
- [write-resource](#), 20
- [zip](#), 7
- zip File Creation, 7
- [zip->output](#), 7
- [zip-verbose](#), 7