

GAP 4 Package IO

Bindings for low level C library I/O routines

Version 1.6

November 2006

Max Neunhöffer

Max Neunhöffer — Email: max.neunhoeffer@math.rwth-aachen.de
— Homepage: <http://www.math.rwth-aachen.de/~Max.Neunhoeffer>
— Address: Lehrstuhl D für Mathematik RWTH Aachen Templer-
graben 64 52062 Aachen Germany

Contents

1	Preface	5
2	Installation of the IO-package	6
3	Functions directly available from the C library	7
3.1	Differences in arguments - an overview	7
3.2	The low-level functions in detail	8
3.2.1	IO_accept	8
3.2.2	IO_bind	8
3.2.3	IO_chdir	8
3.2.4	IO_chmod	9
3.2.5	IO_chown	9
3.2.6	IO_close	9
3.2.7	IO_closedir	9
3.2.8	IO_connect	9
3.2.9	IO_creat	9
3.2.10	IO_dup	9
3.2.11	IO_dup2	10
3.2.12	IO_execv	10
3.2.13	IO_execve	10
3.2.14	IO_execvp	10
3.2.15	IO_exit	10
3.2.16	IO_fchmod	10
3.2.17	IO_fchown	10
3.2.18	IO_fcntl	11
3.2.19	IO_fork	11
3.2.20	IO_fstat	11
3.2.21	IO_gethostbyname	11
3.2.22	IO_getsockopt	11
3.2.23	IO_lchown	11
3.2.24	IO_link	12
3.2.25	IO_listen	12
3.2.26	IO_lseek	12
3.2.27	IO_lstat	12
3.2.28	IO_mkdir	12
3.2.29	IO_mkfifo	12

3.2.30	IO_mknod	12
3.2.31	IO_open	12
3.2.32	IO_opendir	13
3.2.33	IO_pipe	13
3.2.34	IO_read	13
3.2.35	IO_readdir	13
3.2.36	IO_readlink	13
3.2.37	IO_recv	13
3.2.38	IO_recvfrom	13
3.2.39	IO_rename	14
3.2.40	IO_rewinddir	14
3.2.41	IO_rmdir	14
3.2.42	IO_seekdir	14
3.2.43	IO_select	14
3.2.44	IO_send	14
3.2.45	IO_sendto	15
3.2.46	IO_setsockopt	15
3.2.47	IO_socket	15
3.2.48	IO_stat	15
3.2.49	IO_symlink	15
3.2.50	IO_telldir	15
3.2.51	IO_unlink	15
3.2.52	IO_WaitPid	16
3.2.53	IO_write	16
3.3	Further C level functions	16
3.3.1	IO_make_sockaddr_in	16
3.3.2	IO_environ	16
3.3.3	IO_InstallSIGCHLDHandler	16
3.3.4	IO_RestoreSIGCHLDHandler	16
4	Higher level functions for buffered I/O	18
4.1	Types and the creation of File objects	18
4.1.1	IsFile	18
4.1.2	IO_WrapFD	18
4.1.3	IO_File	19
4.2	Reading and writing	19
4.2.1	IO_ReadUntilEOF	19
4.2.2	IO_ReadBlock	19
4.2.3	IO_ReadLine	19
4.2.4	IO_ReadLines	20
4.2.5	IO_HasData	20
4.2.6	IO_Read	20
4.2.7	IO_Write	21
4.2.8	IO_WriteLine	21
4.2.9	IO_WriteLines	21
4.2.10	IO_Flush	21
4.2.11	IO_WriteFlush	22

4.2.12	IO_ReadyForWrite	22
4.2.13	IO_WriteNonBlocking	22
4.2.14	IO_ReadyForFlush	22
4.2.15	IO_FlushNonBlocking	22
4.2.16	IO_Close	23
4.3	Other functions	23
4.3.1	IO_GetFD	23
4.3.2	IO_GetWBuf	23
4.3.3	IO_Select	23
4.3.4	IO_ListDir	23
4.3.5	IO_MakeIPAddressPort	24
4.3.6	IO_Environment	24
4.3.7	IO_MakeEnvList	24
4.4	Inter process communication	24
4.4.1	IO_CloseAllFDs	24
4.4.2	IO_Popen	24
4.4.3	IO_Popen2	25
4.4.4	IO_Popen3	25
4.4.5	IO_SendStringBackground	25
4.4.6	IO_PipeThrough	26
4.4.7	IO_PipeThroughWithError	26
5	Object serialisation (Pickling)	27
5.1	Result objects	27
5.1.1	IO_Error	27
5.1.2	IO_Nothing	27
5.1.3	IO_OK	27
5.2	Pickling and unpickling	28
5.2.1	IO_Pickle	28
5.2.2	IO_Unpickle	28
5.2.3	IO_ClearPickleCache	28
5.3	Extending the pickling framework	28
6	Really random sources	30
6.1	The functions	30
6.1.1	RandomSource	30
7	A client side implementation of the HTTP protocol	31
7.1	Functions for client side HTTP	31
7.1.1	OpenHTTPConnection	31
7.1.2	HTTPRequest	31
7.1.3	HTTPTimeoutForSelect	32
7.1.4	CloseHTTPConnection	32
7.1.5	SingleHTTPRequest	33
8	Examples of usage	34

Chapter 1

Preface

Bindings for low level C library I/O routines

This chapter describes the functions defined in the GAP4 package IO. By default the IO is not automatically loaded by GAP when it is installed. You must load the package with `LoadPackage("IO");` before its functions become available.

Please, send me an e-mail (max.neunhoefffer@math.rwth-aachen.de) if you have any questions, remarks, suggestions, etc. concerning this package. Also, I would like to hear about applications of this package.

Max Neunhöffer

Chapter 2

Installation of the IO-package

To install this package (after extracting the packages archive file to the GAP `pkg` directory) go to the directory `io` (the directory containing this README file) and call

```
./configure [path]
```

where `path` is a path to the main GAP root directory (if not given, the default `../..` is assumed). Afterwards call `make` to compile a binary file.

If you installed GAP on several architectures, you must execute this `configure/make` step on each of the architectures immediately after configuring GAP itself on this architecture.

The package will not work without this step.

The README in the main package directory also explains how to compile the kernel function into a static module.

Chapter 3

Functions directly available from the C library

The following functions from the C library are made available as GAP functions:

`accept`, `bind`, `chdir`, `chmod`, `chown`, `close`, `closedir`, `connect`, `creat`, `dup`, `dup2`, `execv`, `execve`, `execvp`, `exit`, `fchmod`, `fchown`, `fcntl`, `fork`, `fstat`, `gethostbyname`, `getsockopt`, `lchown`, `link`, `listen`, `lseek`, `lstat`, `mkdir`, `mkfifo`, `mknod`, `open`, `opendir`, `pipe`, `read`, `readdir`, `readlink`, `recv`, `recvfrom`, `rename`, `rewinddir`, `rmdir`, `seekdir`, `select`, `send`, `sendto`, `setsockopt`, `socket`, `stat`, `symlink`, `telldir`, `unlink`, `write`.

Use the `man` command in your shell to get information about these functions.

For each of these functions there is a corresponding GAP function in the global record `IO` bound to its name. Apart from minor differences (see below) they take exactly the same arguments as their C counterparts. Strings must be specified as GAP strings and integers as GAP immediate integers. Return values are in general the same as for the C counterparts. However, an error condition is indicated by the value `fail` instead of `-1`, and if the result can only be success or failure, `true` indicates success.

All errors are reported via the `LastSystemError` function.

In the C library a lot of integers are defined as macros in header files. All the necessary values for the above functions are bound to their name in the global `IO` record.

3.1 Differences in arguments - an overview

The `open` function has to be called with three arguments. The version with two arguments is not available on the GAP level.

The `read` function takes four arguments: `fd` is an integer file descriptor, `st` is a GAP string, `offset` is an offset within this string (zero based), and `count` is the maximal number of bytes to read. The data is read and stored into the string `st`, starting at position `offset + 1`. The string `st` is made long enough, such that `count` bytes would fit into it, beginning at position `offset + 1`. The number of bytes read is returned or `fail` in case of an error.

The `write` function is similar, it also takes four arguments: `fd` is an integer file descriptor, `st` is a GAP string, `offset` is an offset within this string (zero based), and `count` is the number of bytes to write, starting from position `offset + 1` in the string `st`. The number of bytes written is returned, or an `fail` in case of an error.

The `opendir` function only returns `true` or `fail`.

The `readdir` function takes no argument. It reads the directory that was specified in the last call to `opendir`. It just returns a string, which is the name of a file or subdirectory in the corresponding directory. It returns `false` after the last file name in the directory or `fail` in case of an error.

The `closedir` function takes no argument. It should be called after `readdir` returned `false` or `fail` to avoid excessive use of file descriptors.

The functions `stat`, `fstat`, and `lstat` only take one argument and return a GAP record that has the same entries as a `struct stat`.

The function `socket` can optionally take a string as third argument. In that case it automatically calls `getprotobyname` to look up the protocol name.

The functions `bind` and `connect` take only one string argument as address field, because the string already encodes the length.

There are two convenience functions `IO_make_sockaddr_in` (3.3.1) and `IO_MakeIPAddressPort` (4.3.5) to create such addresses. The first takes two arguments `addr` and `port`, where `addr` is a string of length 4, containing the 4 bytes of the IP address and `port` is a port number as GAP integer. The function `IO_MakeIPAddressPort` (4.3.5) takes the same arguments, but the first can be a string containing an IP address in dot notation like “137.226.152.77”.

The `setsockopt` function has no argument `optlen`. The length of the string `optval` is taken.

The `select` function works as the function `UNIXSelect` in the GAP library.

As of now, the file locking mechanisms of `fcntl` using `struct flock` are not yet implemented on the GAP level.

3.2 The low-level functions in detail

3.2.1 IO_accept

◇ `IO_accept( fd, addr )` (function)

Accepts an incoming network connection. For details see “man 2 accept”. The argument `addr` can be made with `IO_make_sockaddr_in` (3.3.1) and contains its length such that no third argument is necessary.

3.2.2 IO_bind

◇ `IO_bind( fd, my_addr )` (function)

Binds a local address to a socket. For details see “man 2 bind”. The argument `my_addr` can be made with `IO_make_sockaddr_in` (3.3.1) and contains its length such that no third argument is necessary.

3.2.3 IO_chdir

◇ `IO_chdir( path )` (function)

Changes the current working directory. For details see “man 2 chdir”.

3.2.4 **IO.chmod**

◇ `IO_chmod( pathname, mode )` (function)

Changes the mode of a file. For details see “man 2 chmod”.

3.2.5 **IO.chown**

◇ `IO_chown( path, owner, group )` (function)

Sets owner and/or group of file. For details see “man 2 chown”.

3.2.6 **IO.close**

◇ `IO_close( fd )` (function)

Closes a file descriptor. For details see “man 2 close”.

3.2.7 **IO.closedir**

◇ `IO_closedir( )` (function)

Closes a directory. For details see “man 3 closedir”. Has no arguments, because we only have one `DIR` struct in the C part.

3.2.8 **IO.connect**

◇ `IO_connect( fd, serv_addr )` (function)

Connects to a remote socket. For details see “man 2 connect”. The argument `serv_addr` can be made with `IO_make_sockaddr_in` (3.3.1) and contains its length such that no third argument is necessary.

3.2.9 **IO.creat**

◇ `IO_creat( pathname, mode )` (function)

Creates a new file. For details see “man 2 creat”.

3.2.10 **IO.dup**

◇ `IO_dup( oldfd )` (function)

Duplicates a file descriptor. For details see “man 2 dup”.

3.2.11 **IO_dup2**

◇ `IO_dup2( oldfd, newfd )` (function)

Duplicates a file descriptor to a new one. For details see “man 2 dup2”.

3.2.12 **IO_execv**

◇ `IO_execv( path, argv )` (function)

Replaces the process with another process. For details see “man 3 execv”. The argument `argv` is a list of strings. The called program does not have to be the first argument in this list.

3.2.13 **IO_execve**

◇ `IO_execve( path, argv, envp )` (function)

Replaces the process with another process. For details see “man 3 execve”. The arguments `argv` and `envp` are both lists of strings. The called program does not have to be the first argument in `argv`. The list `envp` can be made with `IO_MakeEnvList` (4.3.7) from a record acquired from `IO_Environment` (4.3.6) and modified later.

3.2.14 **IO_execvp**

◇ `IO_execvp( path, argv )` (function)

Replaces the process with another process. For details see “man 3 execvp”. The argument `argv` is a list of strings. The called program does not have to be the first argument in this list.

3.2.15 **IO_exit**

◇ `IO_exit( status )` (function)

Stops process immediately with return code `status`. For details see “man 2 exit”. The argument `status` must be an integer. Does not return.

3.2.16 **IO_fchmod**

◇ `IO_fchmod( fd, mode )` (function)

Changes mode of an opened file. For details see “man 2 fchmod”.

3.2.17 **IO_fchown**

◇ `IO_fchown( fd, owner, group )` (function)

Changes owner and/or group of an opened file. For details see “man 2 fchown”.

3.2.18 **IO_fcntl**

◇ `IO_fcntl( fd, cmd, arg )` (function)

Does various things to control the behaviour of a file descriptor. For details see “man 2 fcntl”.

3.2.19 **IO_fork**

◇ `IO_fork( )` (function)

Forks off a child process, which is an identical copy. For details see “man 2 fork”. Note that if you want to use the `IO_WaitPid` (3.2.52) function to wait or check for the termination of child processes, you have to activate the `SIGCHLD` handler for this package beforehand by using the function `IO_InstallSIGCHLDHandler` (3.3.3). Note further that after that you cannot use the function `InputOutputLocalProcess` (**Reference: `InputOutputLocalProcess`**) any more, since its `SIGCHLD` handler does not work any more. To switch back to that functionality use the function `IO_RestoreSIGCHLDHandler` (3.3.4).

3.2.20 **IO_fstat**

◇ `IO_fstat( fd )` (function)

Returns the file meta data for an opened file. For details see “man 2 fstat”. A GAP record is returned with the same entries than a `struct stat`.

3.2.21 **IO_gethostbyname**

◇ `IO_gethostbyname( name )` (function)

Return host information by name. For details see “man 3 gethostbyname”. A GAP record is returned with all the relevant information about the host.

3.2.22 **IO_getsockopt**

◇ `IO_getsockopt( fd, level, optname, optval )` (function)

Get a socket option. For details see “man 2 getsockopt”. Note that the argument `optval` carries its length around, such that no 5th argument is necessary.

3.2.23 **IO_lchown**

◇ `IO_lchown( path, owner, group )` (function)

Changes owner and/or group of a file not following links. For details see “man 2 lchown”.

3.2.24 IO_link

◇ `IO_link( oldpath, newpath )` (function)

Create a hard link. For details see “man 2 link”.

3.2.25 IO_listen

◇ `IO_listen( fd, backlog )` (function)

Switch a socket to listening. For details see “man 2 listen”.

3.2.26 IO_lseek

◇ `IO_lseek( fd, offset, whence )` (function)

Seeks with in an open file. For details see “man 2 lseek”.

3.2.27 IO_lstat

◇ `IO_lstat( name )` (function)

Returns the file meta data for a file not following links. For details see “man 2 lstat”. A GAP record is returned with the same entries than a `struct stat`.

3.2.28 IO_mkdir

◇ `IO_mkdir( pathname, mode )` (function)

Creates a directory. For details see “man 2 mkdir”.

3.2.29 IO_mkfifo

◇ `IO_mkfifo( pathname, mode )` (function)

Makes a FIFO special file (a named pipe). For details see “man 3 mkfifo”.

3.2.30 IO_mknod

◇ `IO_mknod( pathname, mode, dev )` (function)

Create a special or ordinary file. For details see “man 2 mknod”.

3.2.31 IO_open

◇ `IO_open( pathname, flags, mode )` (function)

Open and possibly create a file or device. For details see “man 2 open”. Only the variant with 3 arguments can be used.

3.2.32 IO_opendir

◇ `IO_opendir( name )` (function)

Opens a directory. For details see “man 3 opendir”. Note that only `true` is returned if everything is OK, since only one `DIR` struct is stored on the C level and thus only one directory can be open at any time.

3.2.33 IO_pipe

◇ `IO_pipe( )` (function)

Create a pair of file descriptors with a pipe between them. For details see “man 2 pipe”. Note that no arguments are needed. The result is either `fail` in case of an error or a record with two components `toread` and `towrite` bound to the two file descriptors for reading and writing respectively.

3.2.34 IO_read

◇ `IO_read( fd, st, offset, count )` (function)

Reads from file descriptor. For details see “man 2 read”. Note that there is one more argument `offset` to specify at which position in the string `st` the read data should be stored. Note that `offset` zero means at the beginning of the string, which is position 1 in GAP.

3.2.35 IO_readdir

◇ `IO_readdir( )` (function)

Reads from a directory. For details see “man 2 readdir”. Note that no argument is required as we have only one `DIR` struct on the C level.

3.2.36 IO_readlink

◇ `IO_readlink( path, buf, bufsize )` (function)

Reads the value of a symbolic link. For details see “man 2 readlink”.

3.2.37 IO_recv

◇ `IO_recv( fd, st, offset, len, flags )` (function)

Receives data from a socket. For details see “man 2 recv”. Note the additional argument `offset` which plays the same role as for the `IO_read` (3.2.34) function.

3.2.38 IO_recvfrom

◇ `IO_recvfrom( fd, st, offset, len, flags, addr )` (function)

Receives data from a socket with given address. For details see “man 2 recvfrom”. Note the additional argument `offset` which plays the same role as for the `IO_read` (3.2.34) function. The argument `addr` can be made with `IO_make_sockaddr_in` (3.3.1) and contains its length such that no 7th argument is necessary.

3.2.39 `IO_rename`

◇ `IO_rename( oldpath, newpath )` (function)

Renames a file or moves it. For details see “man 2 rename”.

3.2.40 `IO_rewinddir`

◇ `IO_rewinddir( )` (function)

Rewinds a directory. For details see “man 2 rewinddir”. Note that no argument is required as we have only one `DIR` struct on the C level.

3.2.41 `IO_rmdir`

◇ `IO_rmdir( name )` (function)

Removes an empty directory. For details see “man 2 rmdir”.

3.2.42 `IO_seekdir`

◇ `IO_seekdir( offset )` (function)

Sets the position of the next `readdir` call. For details see “man 3 seekdir”. Note that no second argument is required as we have only one `DIR` struct on the C level.

3.2.43 `IO_select`

◇ `IO_select( inlist, outlist, exclist, timeoutsec, timeoutusec )` (function)

Used for I/O multiplexing. For details see “man 2 select”. `inlist`, `outlist` and `exclist` are lists of file descriptors, which are modified. If the corresponding file descriptor is not yet ready, it is replaced by `fail`.

3.2.44 `IO_send`

◇ `IO_send( fd, st, offset, len, flags )` (function)

Sends data to a socket. For details see “man 2 send”. Note that the additional argument `offset` specifies the position of the data to send within the string `st`. It is zero based, meaning that zero indicates the start of the string, which is position 1 in GAP.

3.2.45 **IO_sendto**

◇ `IO_sendto( fd, st, offset, len, flags, addr )` (function)

Sends data to a socket. For details see “man 2 sendto”. Note that the additional argument `offset` specifies the position of the data to send within the string `st`. It is zero based, meaning that zero indicates the start of the string, which is position 1 in GAP. The argument `addr` can be made with `IO_make_sockaddr_in` (3.3.1) and contains its length such that no 7th argument is necessary.

3.2.46 **IO_setsockopt**

◇ `IO_setsockopt( fd, level, optname, optval )` (function)

Sets a socket option. For details see “man 2 setsockopt”. Note that the argument `optval` carries its length around, such that no 5th argument is necessary.

3.2.47 **IO_socket**

◇ `IO_socket( domain, type, protocol )` (function)

Creates a socket, an endpoint for communication. For details see “man 2 socket”. There is one little special: On systems that have `getprotobyname` you can pass a string as third argument `protocol` which is automatically looked up by `getprotobyname`.

3.2.48 **IO_stat**

◇ `IO_stat( pathname )` (function)

Returns the file metadata for the file `pathname`. For details see “man 2 stat”. A GAP record is returned with the same entries than a `struct stat`.

3.2.49 **IO_symlink**

◇ `IO_symlink( oldpath, newpath )` (function)

Creates a symbolic link. For details see “man 2 symlink”.

3.2.50 **IO_telldir**

◇ `IO_telldir( )` (function)

Return current location in directory. For details see “man 3 telldir”. Note that no second argument is required as we have only one `DIR` struct on the C level.

3.2.51 **IO_unlink**

◇ `IO_unlink( pathname )` (function)

Delete a name and possibly the file it refers to. For details see “man 2 unlink”.

3.2.52 **IO.WaitPid**

◇ `IO.WaitPid( pid, wait )` (function)

Waits for the termination of a child process. For details see “man 2 waitpid”. Returns a GAP record describing PID and exit status. The second argument `wait` must be either `true` or `false`. In the first case, the call blocks until new information about a terminated child process is available. In the second case no such waiting is performed, the call returns immediately. See `IO.fork` (3.2.19).

3.2.53 **IO.write**

◇ `IO.write( fd, st, offset, count )` (function)

Writes to a file descriptor. For details see “man 2 write”. Note that the additional argument `offset` specifies the position of the data to send within the string `st`. It is zero based, meaning that zero indicates the start of the string, which is position 1 in GAP.

3.3 Further C level functions

The following functions do not correspond to functions in the C library, but are there to provide convenience to use other functions:

3.3.1 **IO.make_sockaddr_in**

◇ `IO.make_sockaddr_in( ip, port )` (function)

Makes a struct `sockaddr_in` from IP address and port. The IP address must be given as a string of length four, containing the four bytes of an IPv4 address in natural order. The port must be a port number. Returns a string containing the struct, which can be given to all functions above having an address argument.

3.3.2 **IO.environ**

◇ `IO.environ( )` (function)

For details see “man environ”. Returns the current environment as a list of strings of the form “key=value”.

3.3.3 **IO.InstallSIGCHLDHandler**

◇ `IO.InstallSIGCHLDHandler( )` (function)

Installs our SIGCHLD handler. See `IO.fork` (3.2.19).

3.3.4 **IO.RestoreSIGCHLDHandler**

◇ `IO.RestoreSIGCHLDHandler( )` (function)

Restores the original SIGCHLD handler. See `IO_fork` ([3.2.19](#)).

Chapter 4

Higher level functions for buffered I/O

The functions in the previous sections are intended to be a possibility for direct access to the low level I/O functions in the C library. Thus, the calling conventions are strictly as in the original.

The functionality described in this section is implemented completely in the GAP language and is intended to provide a good interface for programming in GAP. The fundamental object for I/O on the C library level is the file descriptor, which is just a non-negative integer representing an open file of the process. The basic idea is to wrap up file descriptors in GAP objects that do the buffering.

Note that considerable care has been taken to ensure that one can do I/O multiplexing with buffered I/O. That is, one always has the possibility to make sure before a read or write operation, that this read or write operation will not block. This is crucial when one wants to serve more than one I/O channel from the same (single-threaded) GAP process. This design principle sometimes made it necessary to have more than one function for a certain operation. Those functions usually differ in a subtle way with respect to their blocking behaviour.

4.1 Types and the creation of File objects

The wrapped file objects are in the following category:

4.1.1 IsFile

◇ `IsFile( o )` (Category)

The category of File objects.

To create objects in this category, one uses the following function:

4.1.2 IO_WrapFD

◇ `IO_WrapFD( fd, rbufsize, wbufsize )` (function)

The argument `fd` must be a file descriptor (i.e. an integer) or -1 (see below).

`rbufsize` can either be `false` for unbuffered reading or an integer buffer size or a string. If it is an integer, a read buffer of that size is used. If it is a string, then `fd` must be -1 and a File object that reads from that string is created.

`wbufsize` can either be `false` for unbuffered writing or an integer buffer size or a string. If it is an integer, a write buffer of that size is used. If it is a string, then `fd` must be `-1` and a `File` object that appends to that string is created.

The result of this function is a new `File` object.

A convenient way to do this for reading or writing of files on disk is the following function:

4.1.3 IO File

```
◇ IO.File( filename[, mode] ) (function)
◇ IO.File( filename[, bufsize] ) (function)
◇ IO.File( filename, mode, bufsize ) (function)
```

The argument `filename` must be a string specifying the path name of the file to work on. `mode` must also be a string with possible values “r”, “w”, or “a”, meaning read access, write access (with creating and truncating), and append access respectively. If `mode` is omitted, it defaults to “r”. `bufsize`, if given, must be a positive integer or `false`, otherwise it defaults to `IO.DefaultBufSize`. Internally, the `IO.open` (3.2.31) function is used and the result file descriptor is wrapped using `IO.WrapFD` (4.1.2) with `bufsize` as the buffer size.

The result is either `fail` in case of an error or a `File` object in case of success.

4.2 Reading and writing

Once a `File` object is created, one can use the following functions on it:

4.2.1 IO_ReadUntilEOF

```
◇ IO.ReadUntilEOF( f ) (function)
```

This function reads all data from the file `f` until the end of file. The data is returned as a GAP string. If the file is already at end of file, an empty string is returned. If an error occurs, then `fail` is returned.

4.2.2 IO_ReadBlock

```
◇ IO.ReadBlock( f, len ) (function)
```

This function gets two arguments, the first argument `f` must be a `File` object and the second argument `len` must be a positive integer. The function tries to read `len` bytes and returns a string of that length. If and only if the end of file is reached earlier, fewer bytes are returned. If an error occurs, `fail` is returned. Note that this function blocks until either `len` bytes are read, or the end of file is reached, or an error occurs.

4.2.3 IO_ReadLine

```
◇ IO.ReadLine( f ) (function)
```

This function gets exactly one argument, which must be a `File` object `f`. It reads one line of data, where the definition of line is operating system dependent. The line end character(s) are included in the result. The function returns a string with the line in case of success and `fail` in case of an error. In the latter case, one can query the error with `LastSystemError` (**Reference: LastSystemError**).

Note that the reading is done via the buffer of `f`, such that this function will be quite fast also for large amounts of data.

If the end of file is hit without a line end, the rest of the file is returned. If the file is already at end of file before the call, then a string of length 0 is returned. Note that this is not an error but the standard end of file convention!

4.2.4 IO.ReadLines

◇ `IO.ReadLines( f[, max] )` (function)

This function gets one or two arguments, the first of which must always be a `File` object `f`. It reads lines of data (where the definition of line is operating system dependent) either until end of file (without a second argument) or up to `max` lines (with a second argument `max`). A list of strings with the result is returned, if everything went well and `fail` otherwise. In the latter case, one can query the error with `LastSystemError` (**Reference: LastSystemError**).

Note that the reading is done via the buffer of `f`, such that this function will be quite fast also for large amounts of data.

If the file is already at the end of file, the function returns a list of length 0. Note that this is not an error but the standard end of file convention!

4.2.5 IO.HasData

◇ `IO.HasData( f )` (function)

This function takes one argument `f` which must be a `File` object. It returns `true` or `false` according to whether there is data to read available in the file `f`. A return value of `true` guarantees that the next call to `IO.Read` (4.2.6) on that file will succeed without blocking and return at least one byte or an empty string to indicate the end of file.

4.2.6 IO.Read

◇ `IO.Read( f, len )` (function)

The function gets two arguments, the first of which must be a `File` object `f`. The second argument must be a positive integer. The function reads data up to `len` bytes. A string with the result is returned, if everything went well and `fail` otherwise. In the latter case, one can query the error with `LastSystemError` (**Reference: LastSystemError**).

Note that the reading is done via the buffer of `f`, such that this function will be quite fast also for large amounts of data.

If the file is already at the end of the file, the function returns a string of length 0. Note that this is not an error!

If a previous call to `IO.HasData` (4.2.5) or to `IO.Select` (4.3.3) indicated that there is data available to read, then it is guaranteed that the function `IO.Read` (4.2.6) does not block and returns at least one byte if the file is not yet at end of file and an empty string otherwise.

4.2.7 `IO.Write`

◇ `IO.Write( f[, things, ...] )` (function)

This function can get an arbitrary number of arguments, the first of which must be a `File` object `f`. All the other arguments are just written to `f` if they are strings. Otherwise, the `String` function is called on them and the result is written out to `f`.

Note that the writing is done buffered. That is, the data is first written to the buffer and only really written out after the buffer is full or after the user explicitly calls `IO.Flush` (4.2.10) on `f`.

The result is either the number of bytes written in case of success or `fail` in case of an error. In the latter case the error can be queried with `LastSystemError` (**Reference: `LastSystemError`**).

Note that this function blocks until all data is at least written into the buffer and might block until data can be sent again if the buffer is full.

4.2.8 `IO.WriteLine`

◇ `IO.WriteLine( f, line )` (function)

Behaves like `IO.Write` (4.2.7) but works on a single string `line` and sends an (operating system dependent) end of line string afterwards. Also `IO.Flush` (4.2.10) is called automatically after the operation, such that one can be sure, that the data is actually written out after the function has completed.

4.2.9 `IO.WriteLines`

◇ `IO.WriteLines( f, list )` (function)

Behaves like `IO.Write` (4.2.7) but works on a list of strings `list` and sends an (operating system dependent) end of line string after each string in the list. Also `IO.Flush` (4.2.10) is called automatically after the operation, such that one can be sure, that the data is actually written out after the function has completed.

4.2.10 `IO.Flush`

◇ `IO.Flush( f )` (function)

This function gets one argument `f`, which must be a `File` object. It writes out all the data that is in the writing buffer. This is not necessary before the call to the function `IO.Close` (4.2.16), since that function calls `IO.Flush` (4.2.10) automatically. However, it is necessary to call `IO.Flush` (4.2.10) after calls to `IO.Write` (4.2.7) to be sure that the data is really sent out. The function returns `true` if everything goes well and `fail` if an error occurs.

Remember that the functions `IO.WriteLine` (4.2.8) and `IO.WriteLines` (4.2.9) implicitly call `IO.Flush` (4.2.10) after they are done.

Note that this function might block until all data is actually written to the file descriptor.

4.2.11 **IO_WriteFlush**

◇ `IO_WriteFlush( f[, things] )` (function)

This function behaves like `IO_Write` (4.2.7) followed by a call to `IO_Flush` (4.2.10). It returns either the number of bytes written or `fail` if an error occurs.

4.2.12 **IO_ReadyForWrite**

◇ `IO_ReadyForWrite( f )` (function)

This function takes one argument `f` which must be a `File` object. It returns `true` or `false` according to whether the file `f` is ready to write. A return value of `true` guarantees that the next call to `IO_WriteNonBlocking` (4.2.13) on that file will succeed without blocking and accept at least one byte.

4.2.13 **IO_WriteNonBlocking**

◇ `IO_WriteNonBlocking( f, st, pos, len )` (function)

This function takes four arguments. The first one `f` must be a `File` object, the second `st` a string, and the arguments `pos` and `len` must be integers, such that positions `pos + 1` until `pos + len` are bound in `st`. The function tries to write up to `len` bytes from `st` from position `pos + 1` to the file `f`. If a previous call to `IO_ReadyForWrite` (4.2.12) or to `IO_Select` (4.3.3) indicates that `f` is writable, then it is guaranteed that the following call to `IO_WriteNonBlocking` (4.2.13) will not block and accept at least one byte of data. Note that it is not guaranteed that all `len` bytes are written. The function returns the number of bytes written or `fail` if an error occurs.

4.2.14 **IO_ReadyForFlush**

◇ `IO_ReadyForFlush( f )` (function)

This function takes one argument `f` which must be a `File` object. It returns `true` or `false` according to whether the file `f` is ready to flush. A return value of `true` guarantees that the next call to `IO_FlushNonBlocking` (4.2.15) on that file will succeed without blocking and flush out at least one byte. Note that this does not guarantee, that this call succeeds to flush out the whole content of the buffer!

4.2.15 **IO_FlushNonBlocking**

◇ `IO_FlushNonBlocking( f )` (function)

This function takes one argument `f` which must be a `File` object. It tries to write all data in the writing buffer to the file descriptor. If this succeeds, the function returns `true` and `false` otherwise. If an error occurs, `fail` is returned. If a previous call to `IO_ReadyForFlush` (4.2.14) or `IO_Select` (4.3.3) indicated that `f` is flushable, then it is guaranteed that the following call to `IO_FlushNonBlocking` (4.2.15) does not block. However, it is not guaranteed that `true` is returned from that call.

4.2.16 IO_Close

◇ `IO_Close( f )` (function)

This function closes the `File` object `f` after writing all data in the write buffer out and closing the file descriptor. All buffers are freed. In case of an error, the function returns `fail` and otherwise `true`.

4.3 Other functions

4.3.1 IO_GetFD

◇ `IO_GetFD( f )` (function)

This function returns the real file descriptor that is behind the `File` object `f`.

4.3.2 IO_GetWBuf

◇ `IO_GetWBuf( f )` (function)

This function gets one argument `f` which must be a `File` object and returns the writing buffer of that `File` object. This is necessary for `File` objects, that are not associated to a real file descriptor but just collect everything that was written in their writing buffer. Remember to use this function before closing the `File` object.

4.3.3 IO_Select

◇ `IO_Select( r, w, f, e, t1, t2 )` (function)

This function is the corresponding function to `IO_select` (3.2.43) for buffered file access. It behaves similarly to that function. The differences are the following: There are four lists of files `r`, `w`, `f`, and `e`. They all can contain either integers (standing for file descriptors) or `File` objects. The list `r` is for checking, whether files or file descriptors are ready to read, the list `w` is for checking whether they are ready to write, the list `f` is for checking whether they are ready to flush, and the list `e` is for checking whether they have exceptions.

For `File` objects it is always first checked, whether there is either data available in a reading buffer or space in a writing buffer. If so, they are immediately reported to be ready. For the remaining files and for all specified file descriptors, the function `IO_select` (3.2.43) is called to get an overview about readiness. The timeout values `t1` and `t2` are set to zero for immediate returning if one of the requested buffers were ready.

`IO_Select` (4.3.3) returns the number of files or file descriptors that are ready to serve or `fail` if an error occurs.

The following function is a convenience function for directory access:

4.3.4 IO_ListDir

◇ `IO_ListDir( pathname )` (function)

This function gets a string containing a path name as single argument and returns a list of strings that are the names of the files in that directory, or `fail`, if an error occurred.

The following function is used to create strings describing a pair of an IP address and a port number in a binary way. These strings can be used in connection with the C library functions `connect`, `bind`, `recvfrom`, and `sendto` for the arguments needing such address pairs.

4.3.5 IO_MakeIPAddressPort

◇ `IO_MakeIPAddressPort( ipstring, portnr )` (function)

This function gets a string `ipstring` containing an IP address in dot notation, i.e. four numbers in the range from 0 to 255 separated by dots “.”, and an integer `portnr`, which is a port number. The result is a string of the correct length to be used for the low level C library functions, wherever IP address port number pairs are needed.

4.3.6 IO_Environment

◇ `IO_Environment( )` (function)

Takes no arguments, uses `IO_environ` (3.3.2) to get the environment and returns a record in which the component names are the names of the environment variables and the values are the values. This can then be changed and the changed record can be given to `IO_MakeEnvList` (4.3.7) to produce again a list which can be used for `IO_execve` (3.2.13) as third argument.

4.3.7 IO_MakeEnvList

◇ `IO_MakeEnvList( r )` (function)

Takes a record as returned by `IO_Environment` (4.3.6) and turns it into a list of strings as needed by `IO_execve` (3.2.13) as third argument.

4.4 Inter process communication

4.4.1 IO_CloseAllFDs

◇ `IO_CloseAllFDs( exceptions )` (function)

Closes all file descriptors except those listed in `exceptions`, which must be a list of integers.

4.4.2 IO_Popen

◇ `IO_Popen( path, argv, mode )` (function)

Starts a child process with either `stdout` or `stdin` being a pipe. The argument `mode` must be either the string “r” or the string “w”.

In the first case, the standard output of the child process will be the writing end of a pipe. A File object for reading connected to the reading end of the pipe is returned. The standard input and standard error of the child process will be the same than the calling GAP process.

In the second case, the standard input of the child process will be the reading end of a pipe. A `File` object for writing connected to the writing end of the pipe is returned. The standard output and standard error of the child process will be the same than the calling GAP process.

In case of an error, `fail` is returned.

The process will usually die, when the pipe is closed, but can also do so without that. It lies in the responsibility of the caller to `IO_WaitPid` (3.2.52) for it, if our `SIGCHLD` handler has been activated (see `IO_InstallSIGCHLDHandler` (3.3.3)).

In either case the `File` object will have the attribute “`ProcessID`” set to the process ID of the child process.

4.4.3 `IO_Popen2`

◇ `IO_Popen2( path, argv )` (function)

A new child process is started. The standard input and standard output of it are pipes. The writing end of the input pipe and the reading end of the output pipe are returned as `File` objects bound to two components “`stdin`” and “`stdout`” (resp.) of the returned record. This means, you have to *write* to “`stdin`” and *read* from “`stdout`” in the calling GAP process. The standard error of the child process will be the same as the one of the calling GAP process.

Returns `fail` if an error occurred.

The process will usually die, when one of the pipes is closed. It lies in the responsibility of the caller to `IO_WaitPid` (3.2.52) for it, if our `SIGCHLD` handler has been activated (see `IO_InstallSIGCHLDHandler` (3.3.3)).

Both `File` objects will have the attribute “`ProcessID`” set to the process ID of the child process, which will also be bound to the “`pid`” component of the returned record.

4.4.4 `IO_Popen3`

◇ `IO_Popen3( path, argv )` (function)

A new child process is started. The standard input, standard output, and standard error of it are pipes. The writing end of the input pipe, the reading end of the output pipe and the reading end of the error pipe are returned as `File` objects bound to two components “`stdin`”, “`stdout`”, and “`stderr`” (resp.) of the returned record. This means, you have to *write* to “`stdin`” and *read* from “`stdout`” and “`stderr`” in the calling GAP process.

Returns `fail` if an error occurred.

The process will usually die, when one of the pipes is closed. It lies in the responsibility of the caller to `IO_WaitPid` (3.2.52) for it, if our `SIGCHLD` handler has been activated (see `IO_InstallSIGCHLDHandler` (3.3.3)).

All three `File` objects will have the attribute “`ProcessID`” set to the process ID of the child process, which will also be bound to the “`pid`” component of the returned record.

4.4.5 `IO_SendStringBackground`

◇ `IO_SendStringBackground( f, st )` (function)

This function uses `IO_Write` (4.2.7) to write the whole string `st` to the `File` object `f`. However, this is done by forking off a child process identical to the calling GAP process that does the sending. The calling GAP process returns immediately, even before anything has been sent away with the result `true`. The forked off sender process terminates itself immediately, after it has sent all data away.

The reason for having this function available is the following: If one uses `IO_Popen2` (4.4.3) or `IO_Popen3` (4.4.4) to start up a child process with standard input and standard output being a pipe, then one usually has the problem, that the child process starts reading some data, but then wants to write data, before it received all data coming. If the calling GAP process would first try to write all data and only start to read the output of the child process after sending away all data, a deadlock situation would occur. This is avoided with the forking and backgrounding approach.

Remember to close the writing end of the standard input pipe in the calling GAP process directly after `IO_SendStringBackground` (4.4.5) has returned, because otherwise the child process might not notice that all data has arrived, because the pipe persists! See the file `popen2.g` in the `example` directory for an example.

Note that with most modern operating systems the forking off of an identical child process does in fact *not* mean a duplication of the total main memory used by both processes, because the operating system kernel will use “copy on write”. However, if a garbage collection happens to become necessary during the sending of the data in the forked off sending process, this might trigger doubled memory usage.

4.4.6 IO_PipeThrough

◇ `IO_PipeThrough( cmd, args, input )` (function)

Returns: a string or fail

Starts the process with the executable stored at the file name `cmd` (must be a string) with arguments in the argument list `args` (a list of strings). The standard input and output of the started process are connected via pipes to the calling process. The content of the string `input` is written to the standard input of the called process and its standard output is read and returned as a string.

All the necessary I/O multiplexing and non-blocking I/O to avoid deadlocks is done in this function.

4.4.7 IO_PipeThroughWithError

◇ `IO_PipeThroughWithError( cmd, args, input )` (function)

Returns: a string or fail

Starts the process with the executable stored at the file name `cmd` (must be a string) with arguments in the argument list `args` (a list of strings). The standard input, output and error of the started process are connected via pipes to the calling process. The content of the string `input` is written to the standard input of the called process and its standard output and error are read and returned as a record with components `out` and `err`, which are strings.

All the necessary I/O multiplexing and non-blocking I/O to avoid deadlocks is done in this function.

Chapter 5

Object serialisation (Pickling)

The idea of “object serialisation” is that one wants to store nearly arbitrary GAP objects to disk or transfer them over the network. To this end, one wants to convert them to a byte stream that is platform independent and can later be converted back to a copy of the same object in memory, be it in the same GAP process or another one maybe even on another machine. The main problem here are the vast amount of different types occurring in GAP and the possibly highly self-referential structure of GAP objects.

The IO package contains a framework to implement object serialisation and implementations for most of the basic data types in GAP. The framework is easily extendible to other types and takes complete care of self-references and corresponding problems. It builds upon the buffered I/O functions described in Section 4. We start by describing the user interface.

5.1 Result objects

The following static objects are used to report about success or failure of the (un-)pickling operations:

5.1.1 IO_Error

◇ `IO_Error` (global variable)

This object is returned if an error occurs.

5.1.2 IO_Nothing

◇ `IO_Nothing` (global variable)

This object is returned when there is nothing to return, for example if an unpickler (see `IO_Unpickle` (5.2.2)) encounters the end of a file.

5.1.3 IO_OK

◇ `IO_OK` (global variable)

This object is returned if everything went well and there is no other canonical value to return to indicate this.

5.2 Pickling and unpickling

The only thing you can do with these special values is to compare them to each other and to other objects.

5.2.1 IO_Pickle

◇ `IO_Pickle( f, ob )` (operation)

Returns: `IO_OK` or `IO_Error`

The argument `f` must be an open, writable `File` object. The object `ob` can be an arbitrary GAP object. The operation “pickles” or “serialises” the object `ob` and writes the result into the `File` object `f`. If everything is OK, the unique value `IO_OK` is returned and otherwise the unique value `IO_Error`. The resulting byte stream can be read again using the operation `IO_Unpickle` (5.2.2) and is platform- and architecture independent. Especially the question whether a system has 32 bit or 64 bit wide words and the question of endianness does not matter.

Note that not all of GAP’s object types are supported but it is relatively easy to extend the system. This package supports in particular boolean values, integers, permutations, rational numbers, finite field elements, cyclotomics, strings, polynomials, rational functions, lists, records, compressed vectors and matrices over finite fields (objects are uncompressed in the byte stream but recompressed during unpickling), and straight line programs.

Self-referential objects built from records and lists are handled correctly and are restored completely with the same self-references during unpickling.

5.2.2 IO_Unpickle

◇ `IO_Unpickle( f )` (operation)

Returns: `IO_Error` or a GAP object

The argument `f` must be an open, readable `File` object. The operation reads from `f` and “unpickles” the next object. If an error occurs, the unique value `IO_Error` is returned. If the `File` object is at end of file, the value `IO_Nothing` is returned. Note that these two values are not picklable, because of their special meaning as return values of this operation here.

5.2.3 IO_ClearPickleCache

◇ `IO_ClearPickleCache( )` (function)

Returns: Nothing

This function clears the “pickle cache”. This cache stores all object pickled in the current recursive call to `IO_Pickle` (5.2.1) and is necessary to handle self-references. Usually it is not necessary to call this function explicitly. Only in the rare case (that should not happen) that a pickling or unpickling operation enters a break loop which is left by the user, the pickle cache has to be cleared explicitly using this function for later calls to `IO_Pickle` (5.2.1) and `IO_Unpickle` (5.2.2) to work!

5.3 Extending the pickling framework

The framework can be extended for other GAP object types as follows:

For pickling, a method for the operation `IO_Pickle` (5.2.1) has to be installed which does the work. If the object to be pickled has subobjects, then the first action of the method is to call the

function `IO.AddToPickled` with the object as argument. This will put it into the pickle cache and take care of self-references. Arbitrary subobjects can then be pickled using recursive calls to the operation `IO.Pickle` (5.2.1) handing down the same `File` object into the recursion. The method must either return `IO.Error` in case of an error or `IO.OK` if everything goes well. Before returning, a method that has called `IO.AddToPickled` must call the function `IO.FinalizePickled` without arguments *under all circumstances*. If this call is missing, global data for the pickling procedure becomes corrupt!

Every pickling method must first write a 4 byte magic value such that later during unpickling of the byte stream the right unpickling method can be called (see below). Then it can write arbitrary data, however, this data should be platform- and architecture independent, and it must be possible to unpickle it later without “lookahead”.

Pickling methods should usually not go into a break loop, because after leaving the user has to call `IO.ClearPickleCache` (5.2.3) explicitly!

Unpickling is implemented as follows: For every 4 byte magic value there must be a function bound to that value in the record `IO.Unpicklers`. If the unpickling operation `IO.Unpickle` (5.2.2) encounters that magic value, it calls the corresponding unpickling function. This function just gets one `File` object as argument. Since the magic value is already read, it can immediately start with reading and rebuilding the serialised object in memory. The method has to take care to restore the object including its type completely.

If an object type has subobjects, the unpickling function has to first create a skeleton of the object without its subobjects, then call `IO.AddToUnpickled` on this skeleton, *before* unpickling subobjects. If things are not done in this order, the handling of self-references down in the recursion will not work! An unpickling function that has called `IO.AddToUnpickled` at the beginning has to call `IO.FinalizeUnpickled` without arguments before returning *under all circumstances*! If this call is missing, global data for the unpickling procedure becomes corrupt!

Of course, unpickling functions can recursively call `IO.Unpickle` (5.2.2) to unpickle subobjects. Apart from this, unpickling functions can use arbitrary reading functions on the `File` object. However, they should only read sequentially and never move the current file position pointer otherwise. An unpickling function should return the newly created object or the value `IO.Error` if an error occurred. They should never go into a break loop, because after leaving the user has to call `IO.ClearPickleCache` (5.2.3) explicitly!

Perhaps the best way to learn how to extend the framework is to study the code for the basic GAP objects in the file `pkg/io/gap/pickle.gi`.

Chapter 6

Really random sources

This section describes so called “real random sources”. It is an extension to the library mechanism of random source objects that uses the devices `/dev/random` and `/dev/urandom` available on Linux systems (and maybe on other operating systems) providing random numbers that are impossible to predict. The idea is that such sources of random numbers are useful to produce unpredictable secret keys for cryptographic applications.

6.1 The functions

6.1.1 RandomSource

◇ `RandomSource( r, dev )` (method)

Returns: a real random source object or `fail`

The first argument `r` must be the GAP filter `IsRealRandomSource` and the second either the string `random` or the string `urandom`. A real random source object is created that draws its random numbers from the kernel devices `/dev/random` and `/dev/urandom` respectively. Whereas `/dev/random` always provides random numbers of not guaranteed “quality”, the device `/dev/urandom` measures its entropy and produces guaranteed unpredictable numbers. However, it might block until enough “random” events (like mouse movements) have been accumulated.

Chapter 7

A client side implementation of the HTTP protocol

The IO contains an implementation of the client side of the HTTP protocol. The basic purpose of this is of course to be able to download data from web servers from the GAP language. However, the HTTP protocol can perform a much bigger variety of tasks.

7.1 Functions for client side HTTP

7.1.1 OpenHTTPConnection

◇ `OpenHTTPConnection( hostname, port )` (function)

Returns: a record

The first argument `hostname` must be a string containing the hostname of the server to connect. The second argument `port` must be an integer in the range from 1 to 65535 and describes the port to connect to on the server.

The function opens a TCP/IP connection to the server and returns a record `conn` with the following components: `conn.sock` is `fail` if an error occurs and otherwise a `File` object linked to the file descriptor of the socket. In case of an error, the component `conn.errormsg` contains an error message, it is otherwise empty. If everything went well then the component `conn.host` is the result from the host name lookup (see `IO_gethostbyname` (3.2.21)) and the component `conn.closed` is set to `false`.

No data is sent or received on the socket in this function.

7.1.2 HTTPRequest

◇ `HTTPRequest( conn, method, uri, header, body, target )` (function)

Returns: a record

This function performs a complete HTTP request. The first argument must be a connection record as returned by a successful call to `OpenHTTPConnection` (7.1.1). The argument `method` must be a valid HTTP request “method” in form of a string. The most common will be `GET`, `POST`, or `HEAD`. The argument `uri` is a string containing the URI of the request, which is given in the first line of the request. This will usually be a relative or absolute path name given to the server. The argument `header` must be a GAP record. Each bound field of this record will be transformed into one header

line with the name of the component being the key and the value the value. All bound values must be strings. The argument `body` must either be a string or `false`. If it is a string, this string is sent away as the body of the request. If no string or an empty string is given, no body will be sent. The header field `Content-Length` is automatically created from the length of the string `body`. Finally, the argument `target` can either be `false` or a string. In the latter case, the body of the request answer is written to the file with the name given in `target`. The `body` component of the result will be the file name in this case. If `target` is `false`, the full body of the answer is stored into the `body` component of the result.

The function sends away the request and awaits the answer. If anything goes wrong during the transfer (for example if the connection is broken prematurely), then the component `statuscode` of the resulting record is 0 and the component `status` is a corresponding error message. In that case, all other fields may or may not be bound to sensible values, according to when the error occurred. If everything goes well, then `statuscode` and `status` are bound to the corresponding values coming from the request answer. `statuscode` is transformed into a GAP integer. The header of the answer is parsed, transformed into a GAP record, and stored into the component `header` of the result. The `body` component of the result record is set as described above. Finally, the `protoversion` component contains the HTTP protocol version number used by the server as a string and the boolean value `closed` indicates, whether or not the function has detected, that the connection has been closed by the server. Note that by default, the connection will stay open, at least for a certain time after the end of the request.

See the description of the global variable `HTTPTimeoutForSelect` (7.1.3) for rules how timeouts are done in this function.

Note that if the `method` is `HEAD`, then no body is expected (none will be sent anyway) and the function returns immediately with empty body. Of course, the `Content-Length` value in the header is as if the request would be done with the `GET` method.

7.1.3 HTTPTimeoutForSelect

◇ `HTTPTimeoutForSelect` (global variable)

This global variable holds a list of length two. By default, both entries are `fail` indicating that `HTTPRequest` (7.1.2) should never timeout and wait forever for an answer. Actually, the two values in this variable are given to the `IO_Select` (4.3.3) function call during I/O multiplexing. That is, the first number is in seconds and the second in milliseconds. Together they lead to a timeout for the HTTP request. If a timeout occurs, an error condition is triggered which returns a record with status code 0 and `status` being the timeout error message.

You can change the timeout by accessing the two entries of this write protected list variable directly.

7.1.4 CloseHTTPConnection

◇ `CloseHTTPConnection( conn )` (function)

Returns: nothing

Closes the connection described by the connection record `conn`. No error can possibly occur.

7.1.5 SingleHTTPRequest

◇ `SingleHTTPRequest( hostname, port, method, uri, header, body, target )` (function)

Returns: a record

The arguments are as the corresponding ones in the functions `OpenHTTPConnection` (7.1.1) and `HTTPRequest` (7.1.2) respectively. This function opens an HTTP connection, tries a single HTTP request and immediately closes the connection again. The result is as for the `HTTPRequest` (7.1.2) function. If an error occurs during the opening of the connection, the `statuscode` value of the result is 0 and the error message is stored in the `status` component of the result.

Chapter 8

Examples of usage

See the `example` directory of the package for some examples of usage. You find there a small server using the TCP/IP protocol and a corresponding client and another small server using the UDP protocol and a corresponding client.

Further, there is an example for the usage of `File` objects, that read from or write to strings.

Finally, there is an example starting up a child process and piping a few megabytes through it using `IO.Popen2` (4.4.3).

Index

IO, [5](#)

CloseHTTPConnection, [32](#)

HTTPRequest, [31](#)

HTTPTimeoutForSelect, [32](#)

IO_accept, [8](#)

IO_bind, [8](#)

IO_chdir, [8](#)

IO_chmod, [9](#)

IO_chown, [9](#)

IO_ClearPickleCache, [28](#)

IO_Close, [23](#)

IO_close, [9](#)

IO_CloseAllFDs, [24](#)

IO_closedir, [9](#)

IO_connect, [9](#)

IO_creat, [9](#)

IO_dup, [9](#)

IO_dup2, [10](#)

IO_environ, [16](#)

IO_Environment, [24](#)

IO_Error, [27](#)

IO_execv, [10](#)

IO_execve, [10](#)

IO_execvp, [10](#)

IO_exit, [10](#)

IO_fchmod, [10](#)

IO_fchown, [10](#)

IO_fcntl, [11](#)

IO_File, [19](#)

IO_Flush, [21](#)

IO_FlushNonBlocking, [22](#)

IO_fork, [11](#)

IO_fstat, [11](#)

IO_GetFD, [23](#)

IO_gethostbyname, [11](#)

IO_getsockopt, [11](#)

IO_GetWBuf, [23](#)

IO_HasData, [20](#)

IO_InstallSIGCHLDHandler, [16](#)

IO_lchown, [11](#)

IO_link, [12](#)

IO_ListDir, [23](#)

IO_listen, [12](#)

IO_lseek, [12](#)

IO_lstat, [12](#)

IO_MakeEnvList, [24](#)

IO_MakeIPAddressPort, [24](#)

IO_make_sockaddr_in, [16](#)

IO_mkdir, [12](#)

IO_mkfifo, [12](#)

IO_mknod, [12](#)

IO_Nothing, [27](#)

IO_OK, [27](#)

IO_open, [12](#)

IO_opendir, [13](#)

IO_Pickle, [28](#)

IO_pipe, [13](#)

IO_PipeThrough, [26](#)

IO_PipeThroughWithError, [26](#)

IO_Popen, [24](#)

IO_Popen2, [25](#)

IO_Popen3, [25](#)

IO_Read, [20](#)

IO_read, [13](#)

IO_ReadBlock, [19](#)

IO_readdir, [13](#)

IO_ReadLine, [19](#)

IO_ReadLines, [20](#)

IO_readlink, [13](#)

IO_ReadUntilEOF, [19](#)

IO_ReadyForFlush, [22](#)

IO_ReadyForWrite, [22](#)

IO_recv, [13](#)

IO_recvfrom, [13](#)

IO.rename, [14](#)

IO_RestoreSIGCHLDHandler, [16](#)

IO_rewinddir, [14](#)
IO_rmdir, [14](#)
IO_seekdir, [14](#)
IO_Select, [23](#)
IO_select, [14](#)
IO_send, [14](#)
IO_SendStringBackground, [25](#)
IO_sendto, [15](#)
IO_setsockopt, [15](#)
IO_socket, [15](#)
IO_stat, [15](#)
IO_symlink, [15](#)
IO_telldir, [15](#)
IO_unlink, [15](#)
IO_Unpickle, [28](#)
IO_WaitPid, [16](#)
IO_WrapFD, [18](#)
IO_Write, [21](#)
IO_write, [16](#)
IO_WriteFlush, [22](#)
IO_WriteLine, [21](#)
IO_WriteLines, [21](#)
IO_WriteNonBlocking, [22](#)
IsFile, [18](#)

OpenHTTPConnection, [31](#)

RandomSource, [30](#)

SingleHTTPRequest, [33](#)