IO

Bindings for low level C library I/O routines

4.9.3

21 June 2025

Max Neunhöffer

Max Neunhöffer Email: max@9hoeffer.de

Copyright

© 2005-2014 by Max Neunhöffer

This package may be distributed under the terms and conditions of the GNU Public License Version 3 or later (at your convenience).

Contents

1 Preface **5**

2 Installation of the IO-package **6**
2.1 Recompiling the documentation 7

3 Functions directly available from the C library **8**
3.1 Differences in arguments - an overview 8
3.2 The low-level functions in detail 9
3.3 Further C level functions 20

4 High level functions for buffered I/O **21**
4.1 Types and the creation of `File` objects 21
4.2 Reading and writing 22
4.3 Other functions 26
4.4 Inter process communication 28

5 Object serialisation (Pickling) **33**
5.1 Result objects 33
5.2 Pickling and unpickling 34
5.3 Extending the pickling framework 34

6 Really random sources **36**
6.1 The functions 36

7 A client side implementation of the HTTP protocol **37**
7.1 Functions for client side HTTP 37

8 Background jobs using fork **40**
8.1 Background jobs 40
8.2 Parallel programming skeletons 42
8.3 Worker farms 44

9 I/O multiplexing **47**
9.1 Introduction 47
9.2 The operations for `IOHub` objects 47
9.3 Examples 50

10 Examples of usage **51**
10.1 Writing and reading a file 51
10.2 Using filtering programs to read and write files 52
10.3 Using filters when reading or writing files sequentially 52
10.4 Accessing a web page 53
10.5 (Un-)Pickling 53

11 License **55**

Index **56**

Chapter 1

Preface

The purpose of this package is to allow efficient and flexible input/output operations from GAP. This is achieved by providing bindings to the low-level I/O functions in the C-library. On top of this an implementation of buffered I/O in the GAP language is provided. Further, a framework for serialisation of arbitrary GAP objects is implemented. Finally, an implementation of the client side of the HTTP protocol is included in the package.

This package allows to use file based I/O, access to links and file systems, pipes, sockets, and the UDP and TCP/IP protocols.

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

For bug reports, feature requests and suggestions, please use our issue tracker.

Chapter 2

Installation of the IO-package

To get the newest version of this GAP 4 package download one of the archive files

- `io-x.x.tar.gz`
- `io-x.x.tar.bz2`
- `io-x.x.zip`

and unpack it using

```
gunzip io-x.x.tar.gz; tar xvf io-x.x.tar
```

or

```
bzip2 -d io-x.x.tar.bz2; tar xvf io-x.x.tar
```

or

```
unzip -x io-x.x.zip
```

respectively.

Do this in a directory called "`pkg`", preferably (but not necessarily) in the "`pkg`" subdirectory of your GAP 4 installation. It creates a subdirectory called "`io`".

The package will not work without the following compilation step.

To compile the C part of the package do (in the `pkg` directory)

```
cd io
./configure
make
```

If you installed the package in another "`pkg`" directory than the standard "`pkg`" directory in your GAP 4 installation, then you have to do two things. Firstly during compilation you have to use the option `--with-gaproot=PATH` of the `configure` script where "PATH" is a path to the main GAP root directory (if not given the default "`../..`" is assumed).

Secondly you have to specify the path to the directory containing your "`pkg`" directory to GAP's list of directories. This can be done by starting GAP with the "`-l`" command line option followed by the name of the directory and a semicolon. Then your directory is prepended to the list of directories searched. Otherwise the package is not found by GAP. Of course, you can add this option to your GAP startup script.

2.1 Recompiling the documentation

Recompiling the documentation is possible by the command "`gap makedoc.g`" in the `io` directory. But this should not be necessary.

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`, `getcwd`, `getenv`, `gethostbyname`, `gethostname`, `getpid`, `getppid`, `getsockname`, `getsockopt`, `gettimeofday`, `gmtime`, `kill`, `lchown`, `link`, `listen`, `localtime`, `lseek`, `lstat`, `mkdir`, `mkfifo`, `mknod`, `mkstemp`, `mkdtemp`, `open`, `opendir`, `pipe`, `read`, `readdir`, `readlink`, `recv`, `recvfrom`, `rename`, `rewinddir`, `rmdir`, `seekdir`, `select`, `send`, `sendto`, `setenv`, `setsockopt`, `socket`, `stat`, `symlink`, `telldir`, `unlink`, `unsetenv`, `write`.

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

For each of these functions there is a corresponding GAP global function with the prefix `IO_` before 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` (**Reference: 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.

Warning: Existence of many of these functions and constants is platform dependent. The compilation process checks existence and this leads to the situation that on the GAP levels the functions and constants are there or not. If you want to develop platform independent GAP code using this package, then you have to check for existence of the functions and constants you need.

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 a `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` (4.3.6) 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` (4.3.6) takes the same arguments, but the first can be a string containing an IP address in dot notation like "137.226.152.77" or a hostname to be looked up.

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

Nearly all of this functions return an integer result in the C library. On the GAP level this is either returned as a non-negative integer in case of success or as `fail` in case of an error (where on the C level −1 would be returned). If the integer can only be 0 for "no error" this is changed to `true` on the GAP level.

3.2.1 IO_accept

▷ `IO_accept(`*fd*`, `*addr*`)` (function)

Returns: an integer or `fail`

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)

Returns: an integer or `fail`

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)

Returns: `true` or `fail`

Changes the current working directory. For details see "`man 2 chdir`".

3.2.4 IO_chmod

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

Returns: `true` or `fail`

Changes the mode of a file. For details see "`man 2 chmod`".

3.2.5 IO_chown

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

Returns: `true` or `fail`

Sets owner and/or group of file. For details see "`man 2 chown`".

3.2.6 IO_close

▷ `IO_close(fd)` (function)

Returns: `true` or `fail`

Closes a file descriptor. For details see "`man 2 close`".

3.2.7 IO_closedir

▷ `IO_closedir()` (function)

Returns: `true` or `fail`

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)

Returns: `true` or `fail`

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)

Returns: an integer or `fail`

Creates a new file. For details see "`man 2 creat`".

3.2.10 IO_dup

▷ `IO_dup(`*`oldfd`*`)` (function)

Returns: an integer or `fail`

Duplicates a file descriptor. For details see "`man 2 dup`".

3.2.11 IO_dup2

▷ `IO_dup2(`*`oldfd, newfd`*`)` (function)

Returns: `true` or `fail`

Duplicates a file descriptor to a new one. For details see "`man 2 dup2`".

3.2.12 IO_execv

▷ `IO_execv(`*`path, argv`*`)` (function)

Returns: `fail` or does not return

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)

Returns: `fail` or does not return

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` (4.3.8) from a record acquired from `IO_Environment` (4.3.7) and modified later.

3.2.14 IO_execvp

▷ `IO_execvp(`*`path, argv`*`)` (function)

Returns: `fail` or does not return

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)

Returns: `true` or `fail`

Changes mode of an opened file. For details see "`man 2 fchmod`".

3.2.17 IO_fchown

▷ `IO_fchown(`*fd*`, `*owner*`, `*group*`)` (function)

Returns: `true` or `fail`

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)

Returns: an integer or `fail`

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)

Returns: an integer or `fail`

Forks off a child process, which is an identical copy. For details see "`man 2 fork`". Note that `IO_fork` activates our SIGCHLD handler (see `IO_InstallSIGCHLDHandler` (3.3.3)). Note that you must use the `IO_WaitPid` (3.2.66) function to wait or check for the termination of child processes, or call `IO_IgnorePid` (3.2.67) to ignore the child.

3.2.20 IO_fstat

▷ `IO_fstat(`*fd*`)` (function)

Returns: a record or `fail`

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_getcwd

▷ `IO_getcwd()` (function)

Returns: a string or `fail`

Returns the current working directory. For details see "`man 3 getcwd`".

3.2.22 IO_getenv

▷ `IO_getenv(`*name*`)` (function)

Returns: a string or `fail`

Return the current value of the environment variable *name*. If the variable is not in the current environment, `fail` is returned. For details see "`man 3 getenv`".

3.2.23 IO_gethostbyname

▷ `IO_gethostbyname(`*`name`*`)` (function)

Returns: a record or `fail`

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

3.2.24 IO_gethostname

▷ `IO_gethostname()` (function)

Returns: a string or `fail`

Return host name. For details see "`man 3 gethostname`".

3.2.25 IO_getpid

▷ `IO_getpid()` (function)

Returns: an integer

Returns the process ID of the current process as an integer. For details see "`man 2 getpid`".

3.2.26 IO_getppid

▷ `IO_getppid()` (function)

Returns: an integer

Returns the process ID of the parent of the current process as an integer. For details see "`man 2 getppid`".

3.2.27 IO_getsockname

▷ `IO_getsockname(`*`fd`*`)` (function)

Returns: a string or `fail`

Get a socket name. For details see "`man 2 getsockname`".

3.2.28 IO_getsockopt

▷ `IO_getsockopt(`*`fd, level, optname, optval`*`)` (function)

Returns: `true` or `false`

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.

3.2.29 IO_gettimeofday

▷ `IO_gettimeofday()` (function)

Returns: A record with components `tv_sec` and `tv_usec`

This returns the time elapsed since 1.1.1970, 0:00 GMT. The component `tv_sec` contains the number of full seconds and the number `tv_usec` the additional microseconds.

3.2.30 IO_gmtime

▷ `IO_gmtime(`*`seconds`*`)` (function)

Returns: A record

The argument is the number of seconds that have elapsed since 1.1.1970, 0:00 GMT. The result is a record with the current Greenwich mean time broken down into date and time as in the C-library function `gmtime`.

3.2.31 IO_kill

▷ `IO_kill(`*`pid, sig`*`)` (function)

Returns: `true` or `fail`

Sends the signal *`sig`* to the process with process ID *`pid`*. For details see "`man 2 kill`". The signal numbers available can be found in the global `IO` record with names like `SIGTERM`.

3.2.32 IO_lchown

▷ `IO_lchown(`*`path, owner, group`*`)` (function)

Returns: `true` or `false`

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

3.2.33 IO_link

▷ `IO_link(`*`oldpath, newpath`*`)` (function)

Returns: `true` or `false`

Create a hard link. For details see "`man 2 link`".

3.2.34 IO_listen

▷ `IO_listen(`*`fd, backlog`*`)` (function)

Returns: `true` or `false`

Switch a socket to listening. For details see "`man 2 listen`".

3.2.35 IO_localtime

▷ `IO_localtime(`*`seconds`*`)` (function)

Returns: A record

The argument is the number of seconds that have elapsed since 1.1.1970, 0:00 GMT. The result is a record with the current local time broken down into date and time as in the C-library function `localtime`.

3.2.36 IO_lseek

▷ `IO_lseek(`*`fd, offset, whence`*`)` (function)

Returns: an integer or `fail`

Seeks within an open file. For details see "`man 2 lseek`".

3.2.37 IO_lstat

▷ `IO_lstat(`*`name`*`)` (function)

Returns: a record or `fail`

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`.

3.2.38 IO_mkdir

▷ `IO_mkdir(`*`pathname, mode`*`)` (function)

Returns: `true` or `false`

Creates a directory. For details see "`man 2 mkdir`".

3.2.39 IO_mkfifo

▷ `IO_mkfifo(`*`pathname, mode`*`)` (function)

Returns: `true` or `false`

Creates a FIFO special file (a named pipe). For details see "`man 3 mkfifo`".

3.2.40 IO_mknod

▷ `IO_mknod(`*`pathname, mode, dev`*`)` (function)

Returns: `true` or `false`

Create a special or ordinary file. For details see "`man 2 mknod`".

3.2.41 IO_mkstemp

▷ `IO_mkstemp(`*`template`*`)` (function)

Returns: an integer or `fail`

Create a special or ordinary file. For details see "`man 3 mkstemp`".

3.2.42 IO_mkdtemp

▷ `IO_mkdtemp(`*`template`*`)` (function)

Returns: a string or `fail`

Create a temporary directory. For details see "`man 3 mkdtemp`".

3.2.43 IO_open

▷ `IO_open(`*`pathname, flags, mode`*`)` (function)

Returns: an integer or `fail`

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

3.2.44 IO_opendir

▷ `IO_opendir(`*`name`*`)` (function)

Returns: `true` or `false`

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.

3.2.45 IO_pipe

▷ `IO_pipe()` (function)

Returns: a record or `fail`

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 filedescriptors for reading and writing respectively.

3.2.46 IO_read

▷ `IO_read(`*`fd, st, offset, count`*`)` (function)

Returns: an integer or `fail`

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. The number of bytes read or `fail` in case of an error is returned.

3.2.47 IO_readdir

▷ `IO_readdir()` (function)

Returns: a string or `fail` or `false`

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. If the directory is read completely `false` is returned, and otherwise a string. An error is indicated by `fail`.

3.2.48 IO_readlink

▷ `IO_readlink(`*`path, buf, bufsize`*`)` (function)

Returns: an integer or `fail`

Reads the value of a symbolic link. For details see "`man 2 readlink`". *`buf`* is modified. The new length of *`buf`* is returned or `fail` in case of an error.

3.2.49 IO_recv

▷ `IO_recv(`*`fd, st, offset, len, flags`*`)` (function)

Returns: an integer or `fail`

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` (3.2.46) function.

3.2.50 IO_recvfrom

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

Returns: an integer or `fail`

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` (3.2.46) 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.

3.2.51 IO_rename

▷ `IO_rename(`*`oldpath, newpath`*`)` (function)

Returns: `true` or `false`

Renames a file or moves it. For details see "`man 2 rename`".

3.2.52 IO_rewinddir

▷ `IO_rewinddir()` (function)

Returns: `true` or `fail`

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. Returns `fail` only, if no prior `IO_opendir` (3.2.44) command has been called.

3.2.53 IO_rmdir

▷ `IO_rmdir(`*`name`*`)` (function)

Returns: `true` or `fail`

Removes an empty directory. For details see "`man 2 rmdir`".

3.2.54 IO_seekdir

▷ `IO_seekdir(`*`offset`*`)` (function)

Returns: `true` or `fail`

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.

3.2.55 IO_select

▷ `IO_select(`*`inlist, outlist, exclist, timeoutsec, timeoutusec`*`)` (function)

Returns: an integer or `fail`

Used for I/O multiplexing. For details see "`man 2 select`". *`inlist`*, *`outlist`* and *`exclist`* are lists of filedescriptors, which are modified. If the corresponding file descriptor is not yet ready, it is replaced by `fail`. The timeout values *`timeoutsec`* and *`timeoutusec`* correspond to the usual arguments of `select`, if both are immediate integers, they are set, otherwise `select` is called with no timeout value.

3.2.56 IO_send

▷ IO_send(*fd, st, offset, len, flags*) (function)

Returns: an integer or fail

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.

3.2.57 IO_sendto

▷ IO_sendto(*fd, st, offset, len, flags, addr*) (function)

Returns: an integer or fail

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.

3.2.58 IO_setenv

▷ IO_setenv(*name, value, overvwrite*) (function)

Returns: true or fail

Set the current value of the environment variable *name* to *value* if it has either not been set before, or *overwrite* is true. For details see "man 3 setenv".

3.2.59 IO_setsockopt

▷ IO_setsockopt(*fd, level, optname, optval*) (function)

Returns: true or fail

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.

3.2.60 IO_socket

▷ IO_socket(*domain, type, protocol*) (function)

Returns: an integer or fail

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.

3.2.61 IO_stat

▷ IO_stat(*pathname*) (function)

Returns: a record or fail

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.

3.2.62 IO_symlink

▷ `IO_symlink(`*`oldpath, newpath`*`)` (function)

Returns: `true` or `fail`

Creates a symbolic link. For details see "`man 2 symlink`".

3.2.63 IO_telldir

▷ `IO_telldir()` (function)

Returns: an integer or `fail`

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.

3.2.64 IO_unlink

▷ `IO_unlink(`*`pathname`*`)` (function)

Returns: `true` or `fail`

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

3.2.65 IO_unsetenv

▷ `IO_unsetenv(`*`name`*`)` (function)

Returns: `true` or `fail`

Remove the environment variable *`name`*. For details see "`man 3 unsetenv`".

3.2.66 IO_WaitPid

▷ `IO_WaitPid(`*`pid, wait`*`)` (function)

Returns: a record or `fail` or `false`

Waits for the termination of a child process. For details see "`man 2 waitpid`". The first argument must be a process id, otherwise the function immediately exits with `fail` as return value.

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. If the child process has not yet terminated, returns `false`; otherwise, returns a GAP record describing the PID, the return value of waitpid, if the process exited normally and the exit status of the process.

See `IO_fork` (3.2.19). If you do not care about the return value of the process, call `IO_IgnorePid` (3.2.67).

3.2.67 IO_IgnorePid

▷ `IO_IgnorePid(`*`pid`*`)` (function)

Returns: Nothing

Disowns a child process. This means there is no need to call `IO_WaitPid` (3.2.66). Calling `IO_WaitPid` (3.2.66) on a pid which was previously passed to `IO_IgnorePid` may cause an infinite loop.F

3.2.68 IO_write

▷ `IO_write(`*`fd, st, offset, count`*`)` (function)

Returns: an integer or `fail`

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)

Returns: a string or `fail`

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)

Returns: a list of strings

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)

Returns: `true` or `false`

Installs our SIGCHLD handler. This functions works as an idempotent. That is, calling it twice does exactly the same as calling it once. It returns `true` when it is called for the first time since then a pointer to the old signal handler is stored in a global variable. This function is automatically called by any function which creates new processes, so never needs to be called unless the handler was explictly disabled with `IO_RestoreSIGCHLDHandler` (3.3.4) See `IO_fork` (3.2.19).

3.3.4 IO_RestoreSIGCHLDHandler

▷ `IO_RestoreSIGCHLDHandler()` (function)

Restores the original SIGCHLD handler. This function works as an idempotent. That is, calling it twice does exactly the same as calling it once. It returns `true` when it is called for the first time after calling `IO_InstallSIGCHLDHandler` (3.3.3). See `IO_fork` (3.2.19).

Chapter 4

High 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.

One remark applies again to nearly all functions presented here: If an error is indicated by the returned value `fail` one can use the library function `LastSystemError` (**Reference: LastSystemError**) to find out more about the cause of the error. This fact is not mentioned with every single function.

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)

Returns: `true` or `false`

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)

Returns: a `File` object

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 (mode)

▷ IO_File(*filename[, mode]*) (function)
▷ IO_File(*filename[, bufsize]*) (function)
▷ IO_File(*filename, mode, bufsize*) (function)

Returns: a File object or fail

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 (3.2.43) 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.

Note that there is a similar function IO_FilteredFile (4.4.9) which also creates a File object but with additional functionality with respect to a pipeline for filtering. It is described in its section in Section 4.4. There is some more low-level functionality to acquire open file descriptors. These can be wrapped into File objects using IO_WrapFD (4.1.2).

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)

Returns: a string or fail

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. Note that you still have to call IO_Close (4.2.16) on the File object to properly close the file later.

4.2.2 IO_ReadBlock

▷ IO_ReadBlock(*f, len*) (function)

Returns: a string or fail

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. For the case of pipes or internet connections it is possible that currently no more data is available, however, by definition the end of file is only reached after the connection has been closed by the other side!

4.2.3 IO_ReadLine

▷ `IO_ReadLine(`*`f`*`)` (function)

Returns: a string or `fail`

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)

Returns: a list of strings or `fail`

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` oterwise. 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)

Returns: `true` or `false`

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)

Returns: a string or `fail`

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` 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)

Returns: an integer or `fail`

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)

Returns: an integer or `fail`

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)

Returns: an integer or `fail`

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)

Returns: `true` or `fail`

This function gets one argument *f*, which must be a `File` object. It writes out all the data that is in the write buffer. This is not necessary before the call to the function `IO_Close` (4.2.16), since

that function calls IO_Flush automatically. However, it is necessary to call IO_Flush 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 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)

Returns: an integer or fail

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)

Returns: true or false

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)

Returns: an integer or fail

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 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)

Returns: true or false

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)

Returns: `true`, `false`, or `fail`

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` does not block. However, it is not guaranteed that `true` is returned from that call.

4.2.16 IO_Close

▷ `IO_Close(f)` (function)

Returns: `true` or `fail`

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`. Note that for pipes to other processes this function collects data about the terminated processes using `IO_WaitPid` (3.2.66).

4.3 Other functions

4.3.1 IO_GetFD

▷ `IO_GetFD(f)` (function)

Returns: an integer

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

4.3.2 IO_GetWBuf

▷ `IO_GetWBuf(f)` (function)

Returns: a string or `false`

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)

Returns: an integer or `fail`

This function is the corresponding function to `IO_select` (3.2.55) 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 (this feature makes the list of File objects to test for flushability necessary). For the remaining files and for all specified file descriptors, the function IO_select (3.2.55) 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 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)

Returns: a list of strings or fail

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.

4.3.5 ChangeDirectoryCurrent

▷ ChangeDirectoryCurrent(*pathname*) (function)

Returns: true on success and fail on failure

Changes the current directory. Returns true on success and fail on failure.

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.

4.3.6 IO_MakeIPAddressPort

▷ IO_MakeIPAddressPort(*ipstring*, *portnr*) (function)

Returns: a string

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. The string *ipstring* can also be a host name, which is then looked up using IO_gethostbyname (3.2.23) to find the IP address.

4.3.7 IO_Environment

▷ IO_Environment() (function)

Returns: a record or fail

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 (4.3.8) to produce again a list which can be used for IO_execve (3.2.13) as third argument.

4.3.8 IO_MakeEnvList

▷ IO_MakeEnvList(*r*) (function)

Returns: a list of strings

Takes a record as returned by `IO_Environment` (4.3.7) 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_FindExecutable

▷ `IO_FindExecutable(`*`path`*`)` (function)

Returns: `fail` or the path to an executable

If the path name *`path`* contains a slash, this function simply checks whether the string *`path`* refers to an executable file. If so, *`path`* is returned as is. Otherwise, `fail` is returned. If the path name *`path`* does not contain a slash, all directories in the environment variable `PATH` are searched for an executable with name *`path`*. If so, the full path to that executable is returned, otherwise `fail`.

This function is used whenever one of the following functions gets an argument that should refer to an executable.

4.4.2 IO_CloseAllFDs

▷ `IO_CloseAllFDs(`*`exceptions`*`)` (function)

Returns: nothing

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

4.4.3 IO_Popen

▷ `IO_Popen(`*`path, argv, mode`*`)` (function)

Returns: a `File` object or `fail`

The argument *`path`* must refer to an executable file in the sense of `IO_FindExecutable` (4.4.1).

Starts a child process using the executable in *`path`* 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 as in 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 as in 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. The `File` object remembers the process ID of the started process and the `IO_Close` (4.2.16) function then calls `IO_WaitPid` (3.2.66) for it to acquire information about the terminated process.

Note that `IO_Popen` activates our SIGCHLD handler (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.

4.4.4 IO_Popen2

▷ `IO_Popen2(`*`path, argv`*`)` (function)

Returns: a record or `fail`

The argument *path* must refer to an executable file in the sense of IO_FindExecutable (4.4.1).

A new child process is started using the executable in *path*. 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. The File objects remember the process ID of the called process and the function call to IO_Close (4.2.16) for the stdout object will call IO_WaitPid (3.2.66) for it to acquire information about the terminated process.

Note that IO_Popen2 activates our SIGCHLD handler (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.

4.4.5 IO_Popen3

▷ IO_Popen3(*path*, *argv*) (function)

Returns: a record or fail

The argument *path* must refer to an executable file in the sense of IO_FindExecutable (4.4.1).

A new child process is started using the executable in *path* 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. All three File objects will remember the process ID of the newly created process and the call to the IO_Close (4.2.16) function for the stdout object will call IO_WaitPid (3.2.66) for it to acquire information about the terminated child process.

Note that IO_Popen3 activates our SIGCHLD handler (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.

4.4.6 IO_StartPipeline

▷ IO_StartPipeline(*progs*, *infd*, *outfd*, *switcherror*) (function)

Returns: a record or fail

The argument *progs* is a list of pairs, the first entry being a path to an executable (in the sense of IO_FindExecutable (4.4.1)), the second an argument list, the argument *infd* is an open file descriptor for reading, *outfd* is an open file descriptor for writing, both can be replaced by the string "open" in which case a new pipe will be opened. The argument *switcherror* is a boolean indicating whether standard error channels are also switched to the corresponding output channels.

This function starts up all processes and connects them with pipes. The input of the first is switched to *infd* and the output of the last to *outfd*.

Returns a record with the following components: `pids` is a list of process ids if everything worked. For each process for which some error occurred the corresponding pid is replaced by `fail`. The `stdin` component is equal to `false`, or to the file descriptor of the writing end of the newly created pipe which is connected to the standard input of the first of the new processes if *`infd`* was "`open`". The `stdout` component is equal to `false` or to the file descriptor of the reading end of the newly created pipe which is connected to the standard output of the last of the new processes if *`outfd`* was "`open`".

Note that the SIGCHLD handler of the IO package is installed by this function (see `IO_InstallSIGCHLDHandler` (3.3.3)) and that it lies in the responsibility of the caller to use `IO_WaitPid` (3.2.66) to ask for the status information of all child processes after their termination, or call `IO_IgnorePid` (3.2.67) to ignore the return value of a process.

4.4.7 IO_StringFilterFile

▷ `IO_StringFilterFile(`*`progs, filename`*`)` (function)

Returns: a string or `fail`

Reads the file with the name *`filename`*, however, a pipeline is created by the processes described by *`progs`* (see `IO_StartPipeline` (4.4.6)) to filter the content of the file through the pipeline. The result is put into a GAP string and returned. If something goes wrong, `fail` is returned.

4.4.8 IO_FileFilterString (append)

▷ `IO_FileFilterString(`*`filename, progs, st[, append]`*`)` (function)

Returns: a string or `fail`

Writes the content of the string *`st`* to the file with the name *`filename`*, however, a pipeline is created by the processes described by *`progs`* (see `IO_StartPipeline` (4.4.6)) to filter the content of the string through the pipeline. The result is put into the file. If the boolean value *`append`* is given and equal to `true`, then the data will be appended to the already existing file. If something goes wrong, `fail` is returned.

4.4.9 IO_FilteredFile

▷ `IO_FilteredFile(`*`progs, filename[, mode][, bufsize]`*`)` (function)

Returns: a `File` object or `fail`

This function is similar to `IO_File` (4.1.3) and behaves nearly identically. The only difference is that a filtering pipeline is switched between the file and the `File` object such that all things read or written respectively are filtered through this pipeline of processes.

The `File` object remembers the started processes and upon the final call to `IO_Close` (4.2.16) automatically uses the `IO_WaitPid` (3.2.66) function to acquire information from the terminated processes in the pipeline after their termination. This means that you do not have to call `IO_WaitPid` (3.2.66) any more after the call to `IO_Close` (4.2.16).

Note that `IO_FilteredFile` activates our SIGCHLD handler (see `IO_InstallSIGCHLDHandler` (3.3.3)).

The `File` object will have the attribute "`ProcessID`" set to the list of process IDs of the child processes.

4.4.10 IO_CompressedFile

▷ `IO_CompressedFile(`*`filename[, mode][, bufsize]`*`)` (function)

Returns: a `File` object or `fail`

This function is a convenience wrapper around `IO_FilteredFile` (4.4.9) which handles a number of common compressed file formats transparently, by calling an external program. The arguments to this function are identical to `IO_File` (4.1.3). If the extension to *`filename`* is one of gz, bz2 or xz, then the file is transparently compressed/uncompressed using gzip, bzip2 or xz respectively. If the extension is none of these, then the command behaves identically to `IO_File` (4.1.3).

Note that as this function calls `IO_FilteredFile` (4.4.9), it will activate our SIGCHLD handler (see `IO_InstallSIGCHLDHandler` (3.3.3)).

When compression / decompression is active, the `File` object will have the attribute "`ProcessID`" set to the list of process IDs of the child processes.

4.4.11 IO_SendStringBackground

▷ `IO_SendStringBackground(`*`f, st`*`)` (function)

This functions 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` (4.4.4) or `IO_Popen3` (4.4.5) 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` 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.

4.4.12 IO_PipeThrough

▷ `IO_PipeThrough(`*`cmd, args, input`*`)` (function)

Returns: a string or `fail`

Starts the process with the executable given by the file name *`cmd`* (in the sense of `IO_FindExecutable` (4.4.1)) 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.

This function properly does `IO_WaitPid` (3.2.66) to wait for the termination of the child process but does not restore the original GAP SIGCHLD signal handler (see `IO_InstallSIGCHLDHandler` (3.3.3)).

4.4.13 IO_PipeThroughWithError

▷ `IO_PipeThroughWithError(`*`cmd, args, input`*`)` (function)

Returns: a record or `fail`

Starts the process with the executable given by the file name *`cmd`* (in the sense of `IO_FindExecutable` (4.4.1)) 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.

This function properly does `IO_WaitPid` (3.2.66) to wait for the termination of the child process but does not restore the original GAP SIGCHLD signal handler (see `IO_InstallSIGCHLDHandler` (3.3.3)).

The functions returns either `fail` if an error occurred, or otherwise a record with components `out` and `err` which are bound to strings containing the full standard output and standard error of the called process, and `status` which is the status returned from the exiting process.

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.

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

5.2 Pickling and unpickling

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 endianess 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/urandom` always provides random numbers of not guaranteed "quality", the device `/dev/random` 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 package 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` (3.2.23)) 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 we 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 it 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.

The previous function allows for a very simple implementation of a function that checks, whether your current GAP installation is up to date:

7.1.6 CheckForUpdates

▷ `CheckForUpdates()` (function)

Returns: nothing

This function has been removed, as it only worked over the insecure HTTP protocol, but not over HTTPS; and the relevant webservice these days only works over HTTPS. If you relied on this functionality, please take a look at the PackageManager package, see https://gap-packages.github.io/PackageManager/.

7.1.7 ReadWeb

▷ `ReadWeb(`*`URL`*`)` (function)

Returns: nothing

This function downloads the file from the given uniform resource locator *URL* using the HTTP protocol and reads the contents into GAP using `Read` (**Reference: Read**).

Note that this can execute arbitrary code on your machine with the privileges of the GAP job running, so you should be very careful what files you download and execute. You have been warned!

Chapter 8

Background jobs using fork

This chapter describes a way to use multi-processor or multi-core machines from within GAP. In its current version the GAP system is a single threaded and single process system. However, modern operating systems allow, via the `fork` system call, to replicate a complete process on the same machine relatively efficiently. That is, at first after a `fork` the two processes actually use the same physical memory such that not much copying needs to be done. The child process is in exactly the same state as the parent process, sharing open files, network connections and the complete status of the workspace. However, whenever a page of memory is written, it is then automatically copied using new, additional physical memory, such that it behaves like a completely separate process. This method is called "copy-on-write".

Thus this is a method to parallelise certain computations. Note however, that from the point of time when the `fork` has occurred, all further communication between the two processes has to be realised via pipes or even files.

The operations and methods described in this chapter help to use GAP in this way and implement certain "skeletons" of parallel programming to make these readily available in GAP. Note that this implementation has its severe limitations and should probably eventually be replaced by a proper multi-threaded version of GAP.

8.1 Background jobs

One creates a background job with the following operation:

8.1.1 BackgroundJobByFork

▷ `BackgroundJobByFork(`*fun, args[, opt]*`)` (operation)

Returns: a background job object or `fail`

This operation creates a background job using `IO_fork` (3.2.19) which starts up as an identical copy of the currently running GAP process. In this child process the function *fun* is called with the argument list *args*. The third argument *opt* must be a record for options. The operation returns either an object representing the background job or `fail` if the startup did not work.

This operation automatically sets up two pipes for communication with the child process. This is in particular used to report the result of the function call to *fun* back to the parent. However, if called without the option `TerminateImmediately` (see below) the child process stays alive even after the completion of *fun* and one can submit further argument lists for subsequent calls to *fun*. Of course,

these additional argument lists will have to be sent over a pipe to the child process. A special case is if the argument *args* is equal to `fail`, in this case the child process is started but does not automatically call *fun* but rather waits in an idle state until an argument list is submitted via the pipe using the `Submit` (8.1.6) operation described below.

There are two components defined which can be bound in the options record *opt*. One is `TerminateImmediately`, if this is bound to `true` then the child process immediately terminates after the function *fun* returns its result. In this case, no pipe for communication from parent to child is created since it would never be used. Note that in this case one can still get the result of the function *fun* using the `Pickup` (8.1.5) operation described below, even when the child has already terminated, since the result is first transmitted back to the parent before termination.

The following operations are available to deal with background job objects:

8.1.2 IsIdle (bgjob)

▷ `IsIdle(`*job*`)` (operation)

Returns: `true`, `false` or `fail`

This operation checks whether or not the background job represented by the object *job* has already finished the function call to its worker function and is now idle. If so, `true` is returned. If it is still running and working on the worker function, `false` is returned. If the background job has already terminated altogether, this operation returns `fail`. Note that if a child process terminates automatically after the first completion of its worker function and sending the result, then the first call to `IsIdle` after completion will return `true` to indicate successful completion and all subsequent calls will return `fail`.

8.1.3 HasTerminated

▷ `HasTerminated(`*job*`)` (operation)

Returns: `true` or `false`

This operation checks whether or not the background job represented by the object *job* has already terminated. If so, `true` is returned, if not, `false` is returned.

8.1.4 WaitUntilIdle

▷ `WaitUntilIdle(`*job*`)` (operation)

Returns: the result of the worker function or `fail`

This operation waits until the worker function of the background job *job* has finished and the job is idle. It then returns the result of the worker function, which has automatically been transmitted to the parent process. If the child process has died before completion `fail` is returned.

8.1.5 Pickup (bgjob)

▷ `Pickup(`*job*`)` (operation)

Returns: the result of the worker function or `fail`

This operation does the same as `WaitUntilIdle` (8.1.4).

8.1.6 Submit (bgjob)

▷ `Submit(job, args)` (operation)

Returns: `true` or `fail`

This submits another argument list *args* for another call to the worker function in the background job *job*. It is an error if either the background job has already terminated or if it is still busy working on the previous argument list. That is, one must only submit another argument in a situation when `IsIdle` (8.1.2) would return `true`. This is for example the case directly after a successful call to `Pickup` (8.1.5) or i `WaitUntilIdle` (8.1.4) which did not return `fail`, unless the background job was created with the `TerminateImmediately` option set to `true`.

This operation returns immediately after submission, when the new argument list has been sent to the child process through a pipe. In particular, it does not await completion of the worker function for the new argument list.

8.1.7 Kill (bgjob)

▷ `Kill(job)` (operation)

Returns: nothing

This kills the background job represented by the object *job* with immediate effect. No more results can be expected from it. Note that unless one has created the background job with the `TerminateImmediately` option set to `true` one always has to call `Kill` on a background job eventually for cleanup purposes. Otherwise, the background job and the connecting pipes remain alive until the parent GAP process terminates.

8.2 Parallel programming skeletons

In this section we document the operations for the available skeletons. For a general description of these ideas see other sources.

8.2.1 ParTakeFirstResultByFork

▷ `ParTakeFirstResultByFork(jobs, args[, opt])` (operation)

Returns: a list of results or `fail`

The argument *jobs* must be a list of GAP functions and the argument *args* a list of the same length containing argument lists with which the job functions can be called. This operation starts up a background job using `fork` for each of the functions in *jobs*, calls it with the corresponding argument list in *args*. As soon as any of the background jobs finishes with a result, `ParTakeFirstResultByFork` terminates all other jobs and reports the results found so far. Note that it can happen that two jobs finish "at the same time" in the sense that both results are received before all other jobs could be terminated. Therefore the result of `ParTakeFirstResultByFork` is a list, in which position *i* is bound if and only if job number *i* returned a result. So in the result at least one entry is bound but it is possible that more than one entry is bound.

You can specify an overall timeout to give up the whole computation if no job finishes by setting the `TimeOut` component of the options record *opt*. In this case you have to set it to a record with two components `tv_sec` and `tv_usec` which are seconds and microseconds respectively, exactly as returned by the `IO_gettimeofday` (3.2.29) function. In the case of timeout an empty list is returned.

8.2.2 ParDoByFork

▷ `ParDoByFork(`*`jobs, args[, opt]`*`)` (operation)

Returns: a list of results or `fail`

The argument *`jobs`* must be a list of GAP functions and the argument *`args`* a list of the same length containing argument lists with which the job functions can be called. This operation starts up a background job using `fork` for each of the functions in *`jobs`*, calls it with the corresponding argument list in *`args`*. As soon as all of the background jobs finish with a result, `ParDoByFork` reports the results found. Therefore the result of `ParDoByFork` is a list, in which position *i* is bound to the result that job number *i* returned.

You can specify an overall timeout to stop the whole computation if not all jobs finish in time by setting the `TimeOut` component of the options record *`opt`*. In this case you have to set it to a record with two components `tv_sec` and `tv_usec` which are seconds and microseconds respectively, exactly as returned by the `IO_gettimeofday` (3.2.29) function. In the case of timeout a list is returned in which the positions corresponding to those jobs that have already finished are bound to the respective results and the other positions are unbound.

8.2.3 ParListByFork

▷ `ParListByFork(`*`l, worker[, opt]`*`)` (operation)

Returns: a list of results or `fail`

This is a parallel version of the `List` (**Reference: list and non-list difference**) function. It applies the function *`worker`* to all elements of the list *`l`* and returns a list containing the results in corresponding positions. You have to specify the component `NumberJobs` in the options record *`opt`* which indicates how many background processes to start. You can optionally use the `TimeOut` option exactly as for `ParDoByFork` (8.2.2), however, if a timeout occurs, `ParListByFork` returns `fail`.

Note that the usefulness of this operation is relatively limited, since every individual result has to be sent back over a pipe from the child process to the parent process. Therefore this only makes sense if the computation time for the worker function dominates the communication time.

8.2.4 ParMapReduceByFork

▷ `ParMapReduceByFork(`*`l, map, reduce[, opt]`*`)` (operation)

Returns: a value or `fail`

This is a parallel version implementation of the classical `MapReduce` pattern. It applies the function *`map`* to all elements of the list *`l`* and then reduces the result using the *`reduce`* function which accepts two return values of *`map`* and returns one of them. Thus, the final result is one return value or `fail` if the startup of the jobs fails. You have to specify the component `NumberJobs` in the options record *`opt`* which indicates how many background processes to start. You can optionally use the `TimeOut` option exactly as for `ParDoByFork` (8.2.2), however, if a timeout occurs, `ParMapReduceByFork` returns `fail`.

Note that this can be very useful because quite often the cumulated computation time for all the worker function calls dominates the communication time for a single result.

8.2.5 IO_CallWithTimeout

▷ `IO_CallWithTimeout(`*`timeout, func, ...`*`)` (function)

▷ `IO_CallWithTimeoutList(`*`timeout, func, arglist`*`)` (function)

IO_CallWithTimeout and IO_CallWithTimeoutList allow calling a function with a limit on length of time it will run. The function is run inside a copy of the current GAP session, so any changes it makes to global variables are thrown away when the function finishes or times out. The return value of *func* is passed back to the current GAP session via IO_Pickle. Note that IO_Pickle may not be available for all objects.

IO_CallWithTimeout is variadic. Any arguments to it beyond the first two are passed as arguments to *func*. IO_CallWithTimeoutList in contrast takes exactly three arguments, of which the third is a list (possibly empty) of arguments to pass to *func*.

If the call completes within the allotted time and returns a value res, the result of IO_CallWithTimeout[List] is a list of the form [true, res].

If the call completes within the allotted time and returns no value, the result of IO_CallWithTimeout[List] is the list [true].

If the call does not complete within the timeout, the result of IO_CallWithTimeout[List] is the list [false]. If the call causes GAP to crash or exit, the result is the list [fail].

The timer is suspended during execution of a break loop and abandoned when you quit from a break loop.

The limit *timeout* is specified as a record. At present the following components are recognised nanoseconds, microseconds, milliseconds, seconds, minutes, hours, days and weeks. Any of these components which is present should be bound to a positive integer, rational or float and the times represented are totalled to give the actual timeout. As a shorthand, a single positive integers may be supplied, and is taken as a number of microseconds. Further components are permitted and ignored, to allow for future functionality.

The precision of the timeouts is not guaranteed, and there is a system dependent upper limit on the timeout which is typically about 8 years on 32 bit systems and about 30 billion years on 64 bit systems. Timeouts longer than this will be reduced to this limit.

Note that the next parallel skeleton is a worker farm which is described in the following section.

8.3 Worker farms

The parallel skeleton of a worker farm is basically nothing but a bunch of background jobs all with the same worker function and all eagerly waiting for work. The only additional concepts needed are an input and an output queue. The input queue contains argument lists and the output queue pairs of argument lists and results.

One creates a worker farm with the following operation:

8.3.1 ParWorkerFarmByFork

▷ ParWorkerFarmByFork(*fun*, *opt*) (operation)

Returns: an object representing the worker farm or fail

This operation creates a worker farm with the worker function *fun* and sets up its input and output queue. An object representing the farm is returned unless not all jobs could be started up in which case fail is returned. After startup all background jobs in the farm are idle. The only valid option in the options record *opt* is NumberJobs and it must be bound to the number of worker jobs in the farm, a positive integer.

The following operations are for worker farm objects:

8.3.2 DoQueues

▷ `DoQueues(wf, block)` (operation)

Returns: nothing

This operation called on a worker farm object *wf* administrates the input and output queues of the worker farm. In particular it checks whether new results are available from the workers and if so it appends them to the output queue. If jobs are idle and the input queue is non-empty, argument lists from the input queue are sent to the idle jobs and removed from the input queue.

This operation must be called regularly to keep up the communication with the clients. It uses `select` and so does not block if the boolean argument *block* is set to `false`. However, if larger chunks of data has to be sent or received this operation might need some time to return.

If the boolean argument *block* is set to true then the `DoQueues` blocks until at least one job has returned a result. This can be used to wait for the termination of all tasks without burning CPU cycles in the parent job. One would repeatedly call `DoQueues` with *block* set to `true` and after each such call check with `IsIdle` (8.3.4) whether all tasks are done. Note that one should no longer call `DoQueues` with *block* set to `true` once this is the case since then it would block forever.

This operation is called automatically by most of the following operations.

8.3.3 Kill (wfarm)

▷ `Kill(wf)` (operation)

Returns: nothing

This operation terminates all background jobs in the farm *wf*, which cannot be used subsequently. One should always call this operation when the worker farm is no longer needed to free resources.

8.3.4 IsIdle (wfarm)

▷ `IsIdle(wf)` (operation)

Returns: `true` or `false`

This operation returns `true` if all background jobs in the worker farm *wf* are idle. This means, that all tasks which have previously been submitted using `Submit` (8.3.5) have been completed and their result been appended to the output queue. The operation `DoQueues` (8.3.2) is automatically called before the execution of `IsIdle`.

8.3.5 Submit (wfarm)

▷ `Submit(wg, arglist)` (operation)

Returns: nothing

This operation submits a task in the form of an argument list for the worker function to the worker farm. It is appended at the end of the input queue. The operation `DoQueues` (8.3.2) is automatically called after the execution of `Submit`, giving the farm a chance to actually send the work out to the worker background jobs.

8.3.6 Pickup (wfarm)

▷ `Pickup(wg, arglist)` (operation)

Returns: nothing

This operation collects all results from the output queue of the worker farm. The output queue is empty after this function returns. The results are reported as a list of pairs, each pair has the input argument list as first component and the output object as second component.

The operation `DoQueues` (8.3.2) is automatically called before the execution of `Pickup`, giving the farm a chance to actually receive some more results from the worker background jobs.

Chapter 9

I/O multiplexing

9.1 Introduction

Whenever one needs to do input/output on more than one connection (file descriptor) at a time, some code is needed to organise the I/O multiplexing. Due to the single-threaded nature of the current GAP language one has to use `IO_select` (3.2.55) and some buffering and queueing to organise this. This chapter describes a relative generic implementation of I/O-multiplexing using so-called `IOHub` (9.2.1) objects. The basic idea is that an `IOHub` (9.2.1) object handles lots of I/O connections at the same time and maintains a buffer for each of them. There is a very simple protocol that marks chunks of data (called "messages") and whenever a message has been received completely it is collected in the input queue of the `IOHub` (9.2.1), marked with the number of the connection it came from. Rather than sending a message away in one go, one would always schedule it for sending by appending it to the output queue. The operation `DoIO` (9.2.13), when called often enough, will then make sure that the message is sent away eventually.

9.2 The operations for `IOHub` objects

In this section, we simply describe the functions and operations to create, use and destroy `IOHub` (9.2.1) objects.

9.2.1 IOHub

▷ `IOHub()` (operation)

Returns: an `IOHub` object

This creates a new `IOHub` object at first without any open connections.

9.2.2 NewConnection

▷ `NewConnection(h, i, o)` (operation)

Returns: a positive integer

This operation adds a new connection to the `IOHub` (9.2.1) object *h*. The arguments *i* and *o* must be Unix file descriptors or 0 and *i* must be open for reading if it is positive and *o* must be open for writing if it is positive. It is allowed that both file descriptors are equal, but they may not both be equal

to 0. The operation returns a positive integer which is the number under which this new connection will be administrated in the IOHub (9.2.1) object. Note that this number is specific to the object *h*.

From the moment these file descriptors are registered with the IOHub (9.2.1) object, every subsequent call to DoIO (9.2.13) will try to do input and output on them. This means in particular that the other side of this connection should be in the same initial state of the protocol. Usually this will be achieved by them being added as a new connection to a corresponding IOHub (9.2.1) object on the other side at the same time.

See also NewTCPConnection (9.2.10) below.

9.2.3 CloseConnection

▷ CloseConnection(*h*, *nr*) (operation)

Returns: nothing

The argument *h* must be an IOHub (9.2.1) object and *nr* the number of a connection which was previously returned by NewConnection (9.2.2). The corresponding connection is closed and removed from the IOHub (9.2.1).

9.2.4 AttachServingSocket

▷ AttachServingSocket(*h*, *addr*, *port*) (operation)

Returns: a Unix file descriptor or fail

The argument *h* must be an IOHub (9.2.1) object, *addr* an IP address or host name as a string and *port* a port number (see also IO_MakeIPAddressPort (4.3.6)). This operation creates a new socket, binds it to the IP address and port and attaches it to the IOHub (9.2.1) object. From this moment on the operation DoIO (9.2.13) will accept new bidirectional TCP/IP connections on that socket and add them to *h*. The operation returns either the file descriptor of the new socket or fail if an error occurred.

9.2.5 ShutdownServingSocket

▷ ShutdownServingSocket(*h*) (operation)

Returns: nothing

The argument *h* must be an IOHub (9.2.1) object. Any server socket which was attached to *h* is shut down, so no new connections will be accepted.

9.2.6 Shutdown

▷ Shutdown(*h*) (operation)

Returns: nothing

The argument *h* must be an IOHub (9.2.1) object. All connections of *h* will be closed using CloseConnection (9.2.3) and any serving socket will be shut down using ShutdownServingSocket (9.2.5). The IOHub (9.2.1) object will not be usable any more after this call.

9.2.7 AcceptNewConnection

▷ AcceptNewConnection(*h*) (operation)

Returns: a positive integer or fail

The argument *h* must be an IOHub (9.2.1) object. The object *h* must have a serving socket attached to it via AttachServingSocket (9.2.4), otherwise fail is returned and nothing happens. One

more connection is accepted through the serving socket. It is added as a new bidirectional TCP/IP connection to the IOHub (9.2.1) object and the new connection number is returned. Note first that this operation blocks until a new connection comes in. Note furthermore that this operation is usually called automatically in DoIO (9.2.13) whenever a new connection has come in, which is reported in the internal IO_select (3.2.55) call. So usually, the client code does not have to call this operation at all.

9.2.8 SubmitOutput

▷ SubmitOutput(*h*, *nr*, *st*) (operation)

Returns: true or fail

The argument *h* must be an IOHub (9.2.1) object, *nr* must be a positive integer which is the number of an open connection of *h* which can be used for output. The argument *st* must be a GAP string. This operation appends the message *st* to the end of the output queue for the connection *nr*. Note that at this stage no output is actually performed automatically. One has to call DoIO (9.2.13) subsequently to actually send the message away.

9.2.9 GetInput

▷ GetInput(*h*, *nr*) (operation)

Returns: a list of length 2

The argument *h* must be an IOHub (9.2.1) object, *nr* must be an integer. If *nr* is positive, this operation returns the earliest message which has come in from connection number *nr* and has not yet been returned by GetInput before. This message is then removed from the input queue. If there is no such message, then false is returned. A message is returned as a plain list of length 2 where the first entry is the connection number it came from and the second entry is a string containing the message itself. If *nr* is equal to 0 then the first message in the input queue from any connection is returned or false if there is no message in the input queue.

9.2.10 NewTCPConnection

▷ NewTCPConnection(*h*, *addr*, *port*) (operation)

Returns: a connection number or fail

The argument *h* must be an IOHub (9.2.1) object, the arguments *addr* and *port* must be an address/port pair as used in IO_MakeIPAddressPort (4.3.6), so *address* can either be a host name or an IP address and *port* is a port number. This operation opens a new TCP connection to the address and port specified, adds a new bidirectional connection to the IOHub (9.2.1) *h* using NewConnection (9.2.2) and returns the connection number specific to the object *h*. If anything goes wrong, fail is returned.

9.2.11 OutputQueue

▷ OutputQueue(*h*) (operation)

Returns: a list

The argument *h* must be an IOHub (9.2.1) object. This returns the internal object for the output queue. Its elements are pairs where the first entry is the connection number where it is going to be sent and the second entry is the message as a string. Only modify this list if you really know what you are doing.

9.2.12 InputQueue

▷ `InputQueue(h)` (operation)

Returns: a list

The argument *h* must be an `IOHub` (9.2.1) object. This returns the internal object for the input queue. Its elements are pairs where the first entry is the connection number from where the message was received and the second entry is the message as a string. Only modify this list if you really know what you are doing.

9.2.13 DoIO

▷ `DoIO(h[, block])` (operation)

Returns: `true` or `false` or `fail`

The argument *h* must be an `IOHub` (9.2.1) object, and the optional second argument *block* must be `true` or `false`. This operation uses `IO_select` (3.2.55) to decide which of the file descriptors belonging to the connections of *h* are ready to read or write. All file descriptors which are ready are served, possibly updating the input and output queues. A possible serving socket is also served accepting a new connection if there is one. The operation `DoIO` loops until no more file descriptors are ready. It returns `true` if some I/O was performed and `false` if not. It returns `fail` if the `IOHub` (9.2.1) is already shut down. The second argument *block* indicates whether or not `DoIO` should block until some I/O has taken place. If this argument is omitted then `false` (non-blocking operation) is the default.

Note that broken connections are silently closed.

9.3 Examples

There is an example hash server in the file `examples/hashserver.g`.

Chapter 10

Examples of usage

For larger examples see the `example` directory of the package. 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.

Another example there shows starting up a child process and piping a few megabytes through it using `IO_Popen2` (4.4.4).

In the following, we present a few explicit, interactive short examples for the usage of the functions in this package. Note that you have to load the IO package with the command `LoadPackage("IO");` before trying these examples.

10.1 Writing and reading a file

The following sequence of commands opens a file with name `guck` and writes some things to it:

Example

```
gap> f := IO_File("guck","w");
<file fd=3 wbufsize=65536 wdata=0>
gap> IO_Write(f,"Hello world\n");
12
gap> IO_WriteLine(f,"Hello world2!");
14
gap> IO_Write(f,12345);
5
gap> IO_Flush(f);
true
gap> IO_Close(f);
true
```

There is nothing special about this, the numbers are numbers of bytes written. Note that only after the `IO_Flush` (4.2.10) command the data is actually written to disk. Before that, it resides in the write buffer of the file. Note further, that the `IO_Flush` (4.2.10) call here would not have been necessary, since the `IO_Close` (4.2.16) call flushes the buffer anyway.

The file can again be read with the following sequence of commands:

Example

```
gap> f := IO_File("guck","r");
<file fd=3 rbufsize=65536 rpos=1 rdata=0>
```

```
gap> IO_Read(f,10);
"Hello worl"
gap> IO_ReadLine(f);
"d\n"
gap> IO_ReadLine(f);
"Hello world2!\n"
gap> IO_ReadLine(f);
"12345"
gap> IO_ReadLine(f);
""
gap> IO_Close(f);
true
```

Note here that reading line-wise can only be done efficiently by using buffered I/O. You can mix calls to `IO_Read` (4.2.6) and to `IO_ReadLine` (4.2.3). The end of file is indicated by an empty string returned by one of the read functions.

10.2 Using filtering programs to read and write files

If you want to write a big amount of data to file you might want to compress it on the fly without using much disk space. This can be achieved with the following command:

Example

```
gap> s := "";; for i in [1..10000] do Append(s,String(i)); od;;
gap> Length(s);
38894
gap> IO_FileFilterString("guck.gz",[["gzip",["-9c"]]],s);
true
gap> sgz := StringFile("guck.gz");;
gap> Length(sgz);
18541
gap> ss := IO_StringFilterFile([["gzip",["-dc"]]],"guck.gz");;
gap> s=ss;
true
```

This sequence of commands needs that the program `gzip` is installed on your system.

10.3 Using filters when reading or writing files sequentially

If you want to process bigger amounts of data you might not want to store all of it in a single GAP string. In that case you might want to access a file on disk sequentially through a filter:

Example

```
gap> f := IO_FilteredFile([["gzip",["-9c"]]],"guck.gz","w");
<file fd=5 wbufsize=65536 wdata=0>
gap> IO_Write(f,"Hello world!\n");
13
gap> IO_Write(f,Elements(SymmetricGroup(5)),"\n");
1359
gap> IO_Close(f);
true
gap> f := IO_FilteredFile([["gzip",["-dc"]]],"guck.gz","r");
```

```
<file fd=4 rbufsize=65536 rpos=1 rdata=0>
gap> IO_ReadLine(f);
"Hello world!\n"
gap> s := IO_ReadLine(f);; Length(s);
1359
gap> IO_Read(f,10);
""
gap> IO_Close(f);
true
```

10.4 Accessing a web page

The IO package has an HTTP client implementation. Using this you can access web pages and other web downloads from within GAP. Here is an example:

Example

```
gap> r := SingleHTTPRequest("www.math.rwth-aachen.de",80,"GET",
>              "/~Max.Neunhoeffer/index.html",rec(),false,false);;
gap> RecNames(r);
[ "protoversion", "statuscode", "status", "header", "body", "closed" ]
gap> r.status;
"OK"
gap> r.statuscode;
200
gap> r.header;
rec( date := "Thu, 07 Dec 2006 22:08:22 GMT",
  server := "Apache/2.0.55 (Ubuntu)",
  last-modified := "Thu, 16 Nov 2006 00:21:44 GMT",
  etag := "\"2179cf-11a5-3c77f600\"", accept-ranges := "bytes",
  content-length := "4517", content-type := "text/html; charset=ISO-8859-1" )
gap> Length(r.body);
4517
```

Of course, the time stamps and exact sizes of the answer may differ when you do this.

10.5 (Un-)Pickling

Assume you have some GAP objects you want to archive to disk grouped together. Then you might do the following:

Example

```
gap> r := rec( a := 1, b := "Max", c := [1,2,3] );
rec( a := 1, b := "Max", c := [ 1, 2, 3 ] )
gap> r.c[4] := r;
rec( a := 1, b := "Max", c := [ 1, 2, 3, ~ ] )
gap> f := IO_File("guck","w");
<file fd=3 wbufsize=65536 wdata=0>
gap> IO_Pickle(f,r);
IO_OK
gap> IO_Pickle(f,[(1,2,3,4),(3,4)]);
IO_OK
```

```
gap> IO_Close(f);
true
```

Then, to read it in again, just do:

Example

```
gap> f := IO_File("guck");
<file fd=3 rbufsize=65536 rpos=1 rdata=0>
gap> IO_Unpickle(f);
rec( a := 1, b := "Max", c := [ 1, 2, 3, ~ ] )
gap> IO_Unpickle(f);
[ (1,2,3,4), (3,4) ]
gap> IO_Unpickle(f);
IO_Nothing
gap> IO_Close(f);
true
```

Note that this works for a certain amount of builtin objects. If you want to archive your own objects or more sophisticated objects you have to use extend the functionality as explained in Section 5.3. However, it works for lists and records and they may be arbitrarily self-referential.

Chapter 11

License

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see https://www.gnu.org/licenses/.

Index

IO, 5

AcceptNewConnection, 48
AttachServingSocket, 48

BackgroundJobByFork, 40

ChangeDirectoryCurrent, 27
CheckForUpdates, 39
CloseConnection, 48
CloseHTTPConnection, 38

DoIO, 50
DoQueues, 45

GetInput, 49

HasTerminated, 41
HTTPRequest, 37
HTTPTimeoutForSelect, 38

InputQueue, 50
IOHub, 47
IO_accept, 9
IO_bind, 10
IO_CallWithTimeout, 43
IO_CallWithTimeoutList, 43
IO_chdir, 10
IO_chmod, 10
IO_chown, 10
IO_ClearPickleCache, 34
IO_Close, 26
IO_close, 10
IO_CloseAllFDs, 28
IO_closedir, 10
IO_CompressedFile, 31
IO_connect, 10
IO_creat, 11
IO_dup, 11
IO_dup2, 11
IO_environ, 20
IO_Environment, 27
IO_Error, 33
IO_execv, 11
IO_execve, 11
IO_execvp, 11
IO_exit, 11
IO_fchmod, 12
IO_fchown, 12
IO_fcntl, 12
IO_File
    bufsize, 22
    mode, 22
    mode and bufsize, 22
IO_FileFilterString
    append, 30
IO_FilteredFile, 30
IO_FindExecutable, 28
IO_Flush, 24
IO_FlushNonBlocking, 26
IO_fork, 12
IO_fstat, 12
IO_getcwd, 12
IO_getenv, 12
IO_GetFD, 26
IO_gethostbyname, 13
IO_gethostname, 13
IO_getpid, 13
IO_getppid, 13
IO_getsockname, 13
IO_getsockopt, 13
IO_gettimeofday, 13
IO_GetWBuf, 26
IO_gmtime, 14
IO_HasData, 23
IO_IgnorePid, 19
IO_InstallSIGCHLDHandler, 20
IO_kill, 14
IO_lchown, 14

IO_link, 14
IO_ListDir, 27
IO_listen, 14
IO_localtime, 14
IO_lseek, 14
IO_lstat, 15
IO_MakeEnvList, 27
IO_MakeIPAddressPort, 27
IO_make_sockaddr_in, 20
IO_mkdir, 15
IO_mkdtemp, 15
IO_mkfifo, 15
IO_mknod, 15
IO_mkstemp, 15
IO_Nothing, 33
IO_OK, 33
IO_open, 15
IO_opendir, 16
IO_Pickle, 34
IO_pipe, 16
IO_PipeThrough, 31
IO_PipeThroughWithError, 32
IO_Popen, 28
IO_Popen2, 28
IO_Popen3, 29
IO_Read, 23
IO_read, 16
IO_ReadBlock, 22
IO_readdir, 16
IO_ReadLine, 23
IO_ReadLines, 23
IO_readlink, 16
IO_ReadUntilEOF, 22
IO_ReadyForFlush, 25
IO_ReadyForWrite, 25
IO_recv, 16
IO_recvfrom, 17
IO_rename, 17
IO_RestoreSIGCHLDHandler, 20
IO_rewinddir, 17
IO_rmdir, 17
IO_seekdir, 17
IO_Select, 26
IO_select, 17
IO_send, 18
IO_SendStringBackground, 31
IO_sendto, 18
IO_setenv, 18
IO_setsockopt, 18
IO_socket, 18
IO_StartPipeline, 29
IO_stat, 18
IO_StringFilterFile, 30
IO_symlink, 19
IO_telldir, 19
IO_unlink, 19
IO_Unpickle, 34
IO_unsetenv, 19
IO_WaitPid, 19
IO_WrapFD, 21
IO_Write, 24
IO_write, 20
IO_WriteFlush, 25
IO_WriteLine, 24
IO_WriteLines, 24
IO_WriteNonBlocking, 25
IsFile, 21
IsIdle
    bgjob, 41
    wfarm, 45

Kill
    bgjob, 42
    wfarm, 45

NewConnection, 47
NewTCPConnection, 49

OpenHTTPConnection, 37
OutputQueue, 49

ParDoByFork, 43
ParListByFork, 43
ParMapReduceByFork, 43
ParTakeFirstResultByFork, 42
ParWorkerFarmByFork, 44
Pickup
    bgjob, 41
    wfarm, 45

RandomSource, 36
ReadWeb, 39

Shutdown, 48

ShutdownServingSocket, 48
SingleHTTPRequest, 39
Submit
    bgjob, 42
    wfarm, 45
SubmitOutput, 49

Timeouts, 43

WaitUntilIdle, 41