SWI-Prolog C-library

Jan Wielemaker
VU University of Amsterdam
The Netherlands
E-mail: J.Wielemaker@vu.nl

July 2, 2018

Abstract

This document describes commonly used foreign language extensions to SWI-Prolog distributed as a package known under the name *clib*. The package defines a number of Prolog libraries with accompagnying foreign libraries.

On Windows systems, the `unix` library can only be used if the whole SWI-Prolog suite is compiled using Cygwin. The other libraries have been ported to native Windows.

Contents

1 Introduction

Many useful facilities offered by one or more of the operating systems supported by SWI-Prolog are not supported by the SWI-Prolog kernel distribution. Including these would enlarge the *footprint* and complicate portability matters while supporting only a limited part of the user-community.

This document describes `unix` to deal with the Unix process API, `socket` to deal with inet-domain TCP and UDP sockets, `cgi` to deal with getting CGI form-data if SWI-Prolog is used as a CGI scripting language, `crypt` to provide password encryption and verification, `sha` providing cryptographic hash functions and `memfile` providing in-memorty pseudo files.

2 library(process): Create processes and redirect I/O

Compatibility SICStus 4
To be done Implement detached option in `process_create/3`

The module `library(process)` implements interaction with child processes and unifies older interfaces such as shell/[1,2], `open(pipe(command), ...)` etc. This library is modelled after SICStus 4.

The main interface is formed by `process_create/3`. If the process id is requested the process must be waited for using `process_wait/2`. Otherwise the process resources are reclaimed automatically.

In addition to the predicates, this module defines a file search path (see `user:file_search_path/2` and `absolute_file_name/3`) named `path` that locates files on the system's search path for executables. E.g. the following finds the executable for `ls`:

```
?- absolute_file_name(path(ls), Path, [access(execute)]).
```

Incompatibilities and current limitations

- Where SICStus distinguishes between an internal process id and the OS process id, this implementation does not make this distinction. This implies that `is_process/1` is incomplete and unreliable.
- SICStus only supports ISO 8859-1 (latin-1). This implementation supports arbitrary OS multi-byte interaction using the default locale.
- It is unclear what the `detached(true)` option is supposed to do. Disable signals in the child? Use `setsid()` to detach from the session? The current implementation uses `setsid()` on Unix systems.
- An extra option `env([Name=Value, ...])` is added to `process_create/3`.

process_create(*+Exe, +Args:list, +Options*) *[det]*

Create a new process running the file *Exe* and using arguments from the given list. *Exe* is a file specification as handed to `absolute_file_name/3`. Typically one use the `path` file alias to specify an executable file on the current PATH. *Args* is a list of arguments that are handed to

the new process. On Unix systems, each element in the list becomes a seperate argument in the new process. In Windows, the arguments are simply concatenated to form the commandline. Each argument itself is either a primitive or a list of primitives. A primitive is either atomic or a term `file(Spec)`. Using `file(Spec)`, the system inserts a filename using the OS filename conventions which is properly quoted if needed.

Options:

stdin(*Spec***)**

stdout(*Spec***)**

stderr(*Spec***)**

Bind the standard streams of the new process. *Spec* is one of the terms below. If `pipe(Pipe)` is used, the Prolog stream is a stream in text-mode using the encoding of the default locale. The encoding can be changed using `set_stream/2`. The options `stdout` and `stderr` may use the same stream, in which case both output streams are connected to the same Prolog stream.

std

Just share with the Prolog I/O streams

null

Bind to a *null* stream. Reading from such a stream returns end-of-file, writing produces no output

pipe(-*Stream***)**

Attach input and/or output to a Prolog stream.

cwd(+*Directory***)**

Run the new process in *Directory*. *Directory* can be a compound specification, which is converted using `absolute_file_name/3`.

env(+*List***)**

Specify the environment for the new process. *List* is a list of Name=Value terms. Note that the current implementation does not pass any environment variables. If unspecified, the environment is inherited from the Prolog process.

process(-*PID***)**

Unify *PID* with the process id of the created process.

detached(+*Bool***)**

In Unix: If `true`, detach the process from the terminal Currently mapped to `setsid();` Also creates a new process group for the child In Windows: If `true`, detach the process from the current job via the CREATE_BREAKAWAY_FROM_JOB flag. In Vista and beyond, processes launched from the shell directly have the 'compatibility assistant' attached to them automatically unless they have a UAC manifest embedded in them. This means that you will get a permission denied error if you try and assign the newly-created PID to a job you create yourself.

window(+*Bool***)**

If `true`, create a window for the process (Windows only)

priority(+*Priority*)

In Unix: specifies the process priority for the newly created process. *Priority* must be an integer between -20 and 19. Positive values are nicer to others, and negative values are less so. The default is zero. Users are free to lower their own priority. Only the super-user may *raise* it to less-than zero.

If the user specifies the `process(-PID)` option, he **must** call `process_wait/2` to reclaim the process. Without this option, the system will wait for completion of the process after the last pipe stream is closed.

If the process is not waited for, it must succeed with status 0. If not, an process_error is raised.

Windows notes

On Windows this call is an interface to the CreateProcess() API. The commandline consists of the basename of *Exe* and the arguments formed from *Args*. Arguments are separated by a single space. If all characters satisfy `iswalnum()` it is unquoted. If the argument contains a double-quote it is quoted using single quotes. If both single and double quotes appear a domain_error is raised, otherwise double-quote are used.

The CreateProcess() API has many options. Currently only the `CREATE_NO_WINDOW` options is supported through the `window(+Bool)` option. If omitted, the default is to use this option if the application has no console. Future versions are likely to support more window specific options and replace `win_exec/2`.

Examples

First, a very simple example that behaves the same as `shell('ls -l')`, except for error handling:

```
?- process_create(path(ls), ['-l'], []).
```

The following example uses grep to find all matching lines in a file.

```
grep(File, Pattern, Lines) :-
        setup_call_cleanup(
            process_create(path(grep), [ Pattern, file(File) ],
                           [ stdout(pipe(Out))
                           ]),
            read_lines(Out, Lines),
            close(Out)).

read_lines(Out, Lines) :-
        read_line_to_codes(Out, Line1),
        read_lines(Line1, Out, Lines).

read_lines(end_of_file, _, []) :- !.
read_lines(Codes, Out, [Line|Lines]) :-
        atom_codes(Line, Codes),
        read_line_to_codes(Out, Line2),
        read_lines(Line2, Out, Lines).
```

Errors `process_error(Exe, Status)` where Status is one of `exit(Code)` or `killed(Signal)`. Raised if the process is waited for (i.e., *Options* does not include `process(-PID)`), and does not exit with status 0.

process_id(*-PID*) *[det]*

True if *PID* is the process id of the running Prolog process.

deprecated Use `current_prolog_flag(pid, PID)`

process_id(*+Process, -PID*) *[det]*

PID is the process id of *Process*. Given that they are united in SWI-Prolog, this is a simple unify.

is_process(*+PID*) *[semidet]*

True if *PID* might be a process. Succeeds for any positive integer.

process_release(*+PID*)

Release process handle. In this implementation this is the same as `process_wait(PID, _)`.

process_wait(*+PID, -Status*) *[det]*
process_wait(*+PID, -Status, +Options*) *[det]*

True if *PID* completed with *Status*. This call normally blocks until the process is finished. *Options*:

timeout(*+Timeout*)

Default: `infinite`. If this option is a number, the waits for a maximum of *Timeout* seconds and unifies *Status* with `timeout` if the process does not terminate within *Timeout*. In this case *PID* is *not* invalidated. On Unix systems only timeout 0 and `infinite` are supported. A 0-value can be used to poll the status of the process.

release(*+Bool*)

Do/do not release the process. We do not support this flag and a domain_error is raised if `release(false)` is provided.

Arguments

Status	is one of `exit(Code)` or `killed(Signal)`, where Code and Signal are integers. If the `timeout` option is used *Status* is unified with `timeout` after the wait timed out.

process_kill(*+PID*) *[det]*
process_kill(*+PID, +Signal*) *[det]*

Send signal to process *PID*. Default is `term`. *Signal* is an integer, Unix signal name (e.g. `SIGSTOP`) or the more Prolog friendly variation one gets after removing `SIG` and downcase the result: `stop`. On Windows systems, *Signal* is ignored and the process is terminated using the TerminateProcess() API. On Windows systems *PID* must be obtained from `process_create/3`, while any *PID* is allowed on Unix systems.

Compatibility SICStus does not accept the prolog friendly version. We choose to do so for compatibility with `on_signal/3`.

process_group_kill(+*PID*) *[det]*
process_group_kill(+*PID, +Signal*) *[det]*

Send signal to the group containing process *PID*. Default is `term`. See `process_wait/1` for a description of signal handling. In Windows, the same restriction on *PID* applies: it must have been created from `process_create/3`, and the the group is terminated via the TerminateJobObject API.

3 library(filesex): Extended operations on files

This module provides additional operations on files. This covers both more obscure and possible non-portable low-level operations and high-level utilities.

Using these Prolog primitives is typically to be preferred over using operating system primitives through `shell/1` or `process_create/3` because (1) there are no potential file name quoting issues, (2) there is no dependency on operating system commands and (3) using the implementations from this library is usually faster.

set_time_file(+*File, -OldTimes, +NewTimes*) *[det]*

Query and set POSIX time attributes of a file. Both *OldTimes* and *NewTimes* are lists of option-terms. Times are represented in SWI-Prolog's standard floating point numbers. New times may be specified as `now` to indicate the current time. Defined options are:

access(*Time*)
Describes the time of last access of the file. This value can be read and written.

modified(*Time*)
Describes the time the contents of the file was last modified. This value can be read and written.

changed(*Time*)
Describes the time the file-structure itself was changed by adding (`link()`) or removing (`unlink()`) names.

Below are some example queries. The first retrieves the access-time, while the second sets the last-modified time to the current time.

```
?- set_time_file(foo, [access(Access)], []).
?- set_time_file(foo, [], [modified(now)]).
```

link_file(+*OldPath, +NewPath, +Type*) *[det]*

Create a link in the filesystem from *NewPath* to *OldPath*. *Type* defines the type of link and is one of `hard` or `symbolic`.

With some limitations, these functions also work on Windows. First of all, the underlying filesystem must support links. This requires NTFS. Second, symbolic links are only supported in Vista and later.

Errors `domain_error(link_type, Type)` if the requested link-type is unknown or not supported on the target OS.

relative_file_name(+*Path:atom, +RelTo:atom, -RelPath:atom*) *[det]*
relative_file_name(-*Path:atom, +RelTo:atom, +RelPath:atom*) *[det]*

True when *RelPath* is *Path*, relative to *RelTo*. *Path* and *RelTo* are first handed to `absolute_file_name/2`, which makes the absolute **and** canonical. Below are two examples:

```
?- relative_file_name('/home/janw/nice',
                      '/home/janw/deep/dir/file', Path).
Path = '../../nice'.

?- relative_file_name(Path, '/home/janw/deep/dir/file', '../../nice').
Path = '/home/janw/nice'.
```

Arguments

All paths must be in canonical POSIX notation, i.e., using / to separate segments in the path. See `prolog_to_os_filename/2`.

bug This predicate is defined as a *syntactical* operation.

directory_file_path(+*Directory, +File, -Path*) *[det]*
directory_file_path(*?Directory, ?File, +Path*) *[det]*

True when *Path* is the full path-name for *File* in Dir. This is comparable to `atom_concat(Directory, File, Path)`, but it ensures there is exactly one / between the two parts. Notes:

- In mode (+,+,-), if *File* is given and absolute, *Path* is unified to *File*.
- Mode (-,-,+) uses `file_directory_name/2` and `file_base_name/2`

copy_file(*From, To*) *[det]*

Copy a file into a new file or directory. The data is copied as binary data.

make_directory_path(+*Dir*) *[det]*

Create *Dir* and all required components (like mkdir -p). Can raise various file-specific exceptions.

copy_directory(+*From, +To*) *[det]*

Copy the contents of the directory *From* to *To* (recursively). If *To* is the name of an existing directory, the *contents* of *From* are copied into *To*. I.e., no subdirectory using the basename of *From* is created.

delete_directory_and_contents(+*Dir*) *[det]*

Recursively remove the directory *Dir* and its contents. If *Dir* is a symbolic link or symbolic links inside *Dir* are encountered, the links are removed rather than their content. Use with care!

delete_directory_contents(+*Dir*) *[det]*

Remove all content from directory *Dir*, without removing *Dir* itself. Similar to `delete_directory_and_contents/2`, if symbolic links are encountered in *Dir*, the links are removed rather than their content.

chmod(*+File, +Spec*) *[det]*

Set the mode of the target file. *Spec* is one of `+Mode`, `-Mode` or a plain *Mode*, which adds new permissions, revokes permissions or sets the exact permissions. *Mode* itself is an integer, a POSIX mode name or a list of POSIX mode names. Defines names are `suid`, `sgid`, `svtx` and the all names defined by the regular expression `[ugo]*[rwx]*`. Specifying none of "ugo" is the same as specifying all of them. For example, to make a file executable for the owner (user) and group, we can use:

```
?- chmod(myfile, +ugx).
```

4 library(uid): User and group management on Unix systems

See also Please check the documentation of your OS for details on the semantics of this predicates.

This module provides and interface to user and group information on Posix systems. In addition, it allows for changing user and group ids. When changing user and group settings for the calling process, bear in mind that:

- Changing user and groups of the calling process requires permission.
- The functions `setgroups()` and `initgroups()` are not part of the POSIX standard and therefore the derived predicates may not be present.

getuid(*-UID*) *[det]*

UID is the real user ID of the calling process.

getgid(*-GID*) *[det]*

GID is the real group ID of the calling process.

geteuid(*-UID*) *[det]*

UID is the effective user ID of the calling process.

getegid(*-GID*) *[det]*

GID is the effective group ID of the calling process.

getgroups(*-GroupsIDs:list(integer)*) *[det]*

GroupsIDs is the set of supplementary group IDs of the calling process. Note that these are numeric identifiers. Use `group_info/2` to obtain details on the returned group identifiers.

user_info(*+User, -UserData*) *[det]*

UserData represent the passwd information for *User*. *User* is either a numeric UID or a user name. The predicate `user_data/3` can be used to extract information from *UserData*.

user_data(*?Field, ?UserData, ?Value*)

Value is the value for *Field* in *UserData*. Defined fields are:

name
Name of the user

password
Password hash of the user (or `x` if this is not accessible)

uid
Numeric user id of the user

gid
Numeric primary group id of the user

comment
The *gecos* field

home
Home directory of the user

shell
Default (login) shell of the user.

group_info(+*Group, -GroupData*) *[det]*
GroupData represent the group information for *Group*. *Group* is either a numeric GID or a group name. The predicate `group_data/3` can be used to extract information from *GroupData*.

group_data(*?Field, ?GroupData, ?Value*)
Value is the value for *Field GroupData*. Defined fields are:

name
Name of the user

password
Password hash of the user (or `x` if this is not accessible)

gid
Numeric group id of the group

members
List of user-names that are member of this group.

setuid(+*UID*)
Set the user id of the calling process.

seteuid(+*UID*)
Set the effective user id of the calling process.

setgid(+*GID*)
Set the group id of the calling process.

setegid(+*GID*)
Set the effective group id of the calling process.

initgroups(+*User, +Group*) *[det]*
Initialise the group access list of the calling process to the registered groups for *User* and the group *Group*. This predicate is only available if the underlying OS provides it.

setgroups(+*Groups:list(integer)*) *[det]*

Set the group access list of the caling process to the indicated groups. This predicate is only available if the underlying OS provides it.

set_user_and_group(+*User*) *[det]*

set_user_and_group(+*User, +Group*) *[det]*

Set the UID and GID to the *User*. *User* is either a UID or a user name. If *Group* is not specified, the primary group of *User* is used. If `initgroups/2` is available, the resulting group access list of the calling process consists of the registered groups for *User* and the specified *Group*.

5 library(syslog): Unix syslog interface

See also

- `detach_IO/1` to detach normal I/O of the process and remove it from the process group.
- `fork/1` to create a daemon process.
- `library(uid)` to manage user identifiers (e.g., drop root privileges).

This library provides an interface to the Unix `syslog()` facility. The interface is an almost direct translation of the POSIX syslog API, with two additions:

- `syslog/3` exploits `format/3` to format syslog messages
- The library integrates into `library(debug)` using `prolog:debug_print_hook/3`, where debug *topics* are mapped to syslog *priorities* and remaining debug *topics* are mapped to the syslog *priority* `debug`.

Note that this interface makes no attempt to abstract over logging facilities of operating systems. We expect that such abstractions will be implemented at the Prolog level using multiple integrations into `library(debug)`.

openlog(+*Ident:atom, +Options:list(atom), +Facility:atom*) *[det]*

Open system log. This predicate provides a direct interface into the `openlog()` library call. If the library call is successful, it runs `at_halt(closelog)` to ensure closing the system log on clean exit.

Arguments

Ident	prepended to every message, and is typically set to the program name.
Options	is a list of options. Values are corresponding C options, after removing =LOG_= and translation to lower case: `cons`, `ndelay`, `nowait`, `odelay`, `perror`, `pid`.
Facility	is one of `auth`, `authpriv`, `cron`, `daemon`, `ftp`, `kern`, `local0` ... `local7`, `lpr`, `mail`, `news`, `syslog`, `user` or `uucp`.

syslog(+*Priority, +Message*) *[det]*

Send a message to the system log. Note that `syslog/2` implicitly opens a connection to the system log if such a connection has not been opened explicitly using `openlog/3`.

Arguments

Priority is one of `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info` or `debug`.

syslog(+*Priority, +Format, +Args*) *[det]*

Send a formatted message to the system log if system logging is opened using `openlog/3`. This predicate combined `format/3` with `syslog/2`. If there is no open syslog connection, `syslog/3` calls `print_message/2`.

closelog *[det]*

Close the system log.

prolog:debug_print_hook(+*Topic, +Format, +Args*) *[semidet,multifile]*

Integration of `debug/3` with the syslog facility. If syslog is enabled, `debug/3` is re-routed to use the syslog facilities. If the *topic* of the debug message matches one of the sylog *priority* values (see `syslog/2`), the message is sent with the corresponding syslog priority. Otherwise it it sent with the `debug` priority.

6 library(socket): Network socket (TCP and UDP) library

The `library(socket)` provides TCP and UDP inet-domain sockets from SWI-Prolog, both client and server-side communication. The interface of this library is very close to the Unix socket interface, also supported by the MS-Windows *winsock* API. SWI-Prolog applications that wish to communicate with multiple sources have three options:

- Use I/O multiplexing based on `wait_for_input/3`. On Windows systems this can only be used for sockets, not for general (device-) file handles.
- Use multiple threads, handling either a single blocking socket or a pool using I/O multiplexing as above.
- Using XPCE's class `socket` which synchronises socket events in the GUI event-loop.

6.1 Client applications

Using this library to establish a TCP connection to a server is as simple as opening a file. See also `http_open/3`.

```
dump_swi_homepage :-
    setup_call_cleanup(
        tcp_connect(www.swi-prolog.org:http, Stream, []),
        ( format(Stream,
                 'GET / HTTP/1.1~n\c
                  Host: www.swi-prolog.org~n\c
                  Connection: close~n~n', []),
          flush_output(Stream),
          copy_stream_data(Stream, current_output)
        ),
        close(S)).
```

To deal with timeouts and multiple connections, threads, `wait_for_input/3` and/or non-blocking streams (see `tcp_fcntl/3`) can be used.

6.2 Server applications

The typical sequence for generating a server application is given below. To close the server, use `close/1` on *AcceptFd*.

```
create_server(Port) :-
      tcp_socket(Socket),
      tcp_bind(Socket, Port),
      tcp_listen(Socket, 5),
      tcp_open_socket(Socket, AcceptFd, _),
      <dispatch>
```

There are various options for <dispatch>. The most commonly used option is to start a Prolog thread to handle the connection. Alternatively, input from multiple clients can be handled in a single thread by listening to these clients using `wait_for_input/3`. Finally, on Unix systems, we can use `fork/1` to handle the connection in a new process. Note that `fork/1` and threads do not cooperate well. Combinations can be realised but require good understanding of POSIX thread and fork-semantics.

Below is the typical example using a thread. Note the use of `setup_call_cleanup/3` to guarantee that all resources are reclaimed, also in case of failure or exceptions.

```
dispatch(AcceptFd) :-
        tcp_accept(AcceptFd, Socket, _Peer),
        thread_create(process_client(Socket, Peer), _,
                      [ detached(true)
                      ]),
        dispatch(AcceptFd).

process_client(Socket, Peer) :-
        setup_call_cleanup(
            tcp_open_socket(Socket, StreamPair),
            handle_service(In, StreamPair),
            close(StreamPair)).

handle_service(StreamPair) :-
        ...
```

6.3 TCP socket predicates

tcp_socket(*-SocketId*) *[det]*

Creates an INET-domain stream-socket and unifies an identifier to it with *SocketId*. On MS-Windows, if the socket library is not yet initialised, this will also initialise the library.

tcp_close_socket(+*SocketId*) *[det]*

Closes the indicated socket, making *SocketId* invalid. Normally, sockets are closed by closing both stream handles returned by `open_socket/3`. There are two cases where `tcp_close_socket/1` is used because there are no stream-handles:

- If, after `tcp_accept/3`, the server uses `fork/1` to handle the client in a sub-process. In this case the accepted socket is not longer needed from the main server and must be discarded using `tcp_close_socket/1`.
- If, after discovering the connecting client with `tcp_accept/3`, the server does not want to accept the connection, it should discard the accepted socket immediately using `tcp_close_socket/1`.

tcp_open_socket(+*SocketId, -StreamPair*) *[det]*

Create streams to communicate to *SocketId*. If *SocketId* is a master socket (see `tcp_bind/2`), *StreamPair* should be used for `tcp_accept/3`. If *SocketId* is a connected (see `tcp_connect/2`) or accepted socket (see `tcp_accept/3`), *StreamPair* is unified to a stream pair (see `stream_pair/3`) that can be used for reading and writing. The stream or pair must be closed with `close/1`, which also closes *SocketId*.

tcp_open_socket(+*SocketId, -InStream, -OutStream*) *[det]*

Similar to `tcp_open_socket/2`, but creates two separate sockets where `tcp_open_socket/2` would have created a stream pair.

> **deprecated** New code should use `tcp_open_socket/2` because closing a stream pair is much easier to perform safely.

tcp_bind(*SocketId, ?Address*) *[det]*

Bind the socket to *Address* on the current machine. This operation, together with `tcp_listen/2` and `tcp_accept/3` implement the *server-side* of the socket interface. *Address* is either an plain *Port* or a term HostPort. The first form binds the socket to the given port on all interfaces, while the second only binds to the matching interface. A typical example is below, causing the socket to listen only on port 8080 on the local machine's network.

```
tcp_bind(Socket, localhost:8080)
```

If *Port* is unbound, the system picks an arbitrary free port and unifies *Port* with the selected port number. *Port* is either an integer or the name of a registered service. See also `tcp_connect/4`.

tcp_listen(+*SocketId, +BackLog*) *[det]*

Tells, after `tcp_bind/2`, the socket to listen for incoming requests for connections. Backlog indicates how many pending connection requests are allowed. Pending requests are requests that are not yet acknowledged using `tcp_accept/3`. If the indicated number is exceeded, the requesting client will be signalled that the service is currently not available. A commonly used default value for Backlog is 5.

tcp_accept(+*Socket, -Slave, -Peer*) *[det]*

This predicate waits on a server socket for a connection request by a client. On success, it creates a new socket for the client and binds the identifier to *Slave*. *Peer* is bound to the IP-address of the client.

tcp_connect(+*SocketId, +HostAndPort*) *[det]*

Connect *SocketId*. After successful completion, `tcp_open_socket/3` can be used to create I/O-Streams to the remote socket. This predicate is part of the low level client API. A connection to a particular host and port is realised using these steps:

```
    tcp_socket(Socket),
    tcp_connect(Socket, Host:Port),
    tcp_open_socket(Socket, StreamPair)
```

Typical client applications should use the high level interface provided by `tcp_connect/3` which avoids resource leaking if a step in the process fails, and can be hooked to support proxies. For example:

```
    setup_call_cleanup(
        tcp_connect(Host:Port, StreamPair, []),
        talk(StreamPair),
        close(StreamPair))
```

tcp_connect(+*Socket, +Address, -Read, -Write*) *[det]*

Connect a (client) socket to *Address* and return a bi-directional connection through the stream-handles *Read* and *Write*. This predicate may be hooked by defining `socket:tcp_connect_hook/4` with the same signature. Hooking can be used to deal with proxy connections. E.g.,

```
:- multifile socket:tcp_connect_hook/4.

socket:tcp_connect_hook(Socket, Address, Read, Write) :-
    proxy(ProxyAdress),
    tcp_connect(Socket, ProxyAdress),
    tcp_open_socket(Socket, Read, Write),
    proxy_connect(Address, Read, Write).
```

deprecated New code should use `tcp_connect/3` called as `tcp_connect(+Address, -StreamPair, +Options)`.

tcp_connect(+*Address, -StreamPair, +Options*) *[det]*

tcp_connect(+*Socket, +Address, -StreamPair*) *[det]*

Establish a TCP communication as a client. The +,-,+ mode is the preferred way for a client to establish a connection. This predicate can be hooked to support network proxies. To use a proxy, the hook `proxy_for_url/3` must be defined. Permitted options are:

bypass_proxy(+*Boolean*)
Defaults to `false`. If `true`, do not attempt to use any proxies to obtain the connection

nodelay(+*Boolean*)
Defaults to `false`. If `true`, set nodelay on the resulting socket using `tcp_setopt(Socket, nodelay)`

The +,+,- mode is deprecated and does not support proxies. It behaves like `tcp_connect/4`, but creates a stream pair (see `stream_pair/3`).

Errors `proxy_error(tried(ResultList))` is raised by mode (+,-,+) if proxies are defines by `proxy_for_url/3` but no proxy can establsh the connection. *ResultList* contains one or more terms of the form `false(Proxy)` for a hook that simply failed or `error(Proxy, ErrorTerm)` for a hook that raised an exception.
See also `library(http/http_proxy)` defines a hook that allows to connect through HTTP proxies that support the `CONNECT` method.

tcp_select(+*ListOfStreams, -ReadyList, +TimeOut*)
Same as the built-in `wait_for_input/3`, but integrates better with event processing and the various options of sockets for Windows. On non-windows systems this simply calls `wait_for_input/3`.

try_proxy(+*Proxy, +TargetAddress, -Socket, -StreamPair*) *[semidet,multifile]*
Attempt a socket-level connection via the given proxy to *TargetAddress*. The *Proxy* argument must match the output argument of `proxy_for_url/3`. The predicate `tcp_connect/3` (and `http_open/3` from the `library(http/http_open)`) collect the results of failed proxies and raise an exception no proxy is capable of realizing the connection.

The default implementation recognises the values for *Proxy* described below. The `library(http/http_proxy)` adds `proxy(Host,Port)` which allows for HTTP proxies using the `CONNECT` method.

direct
Do not use any proxy

socks(*Host, Port*)
Use a SOCKS5 proxy

proxy_for_url(+*URL, +Hostname, -Proxy*) *[nondet,multifile]*
This hook can be implemented to return a proxy to try when connecting to *URL*. Returned proxies are tried in the order in which they are returned by the multifile hook `try_proxy/4`. Pre-defined proxy methods are:

direct
connect directly to the resource

proxy(*Host, Port*)
Connect to the resource using an HTTP proxy. If the resource is not an HTTP *URL*, then try to connect using the CONNECT verb, otherwise, use the GET verb.

socks(*Host, Port*)
Connect to the resource via a SOCKS5 proxy

These correspond to the proxy methods defined by PAC *Proxy* auto-config. Additional methods can be returned if suitable clauses for `http:http_connection_over_proxy/6` or `try_proxy/4` are defined.

tcp_setopt(+*SocketId, +Option*) *[det]*

Set options on the socket. Defined options are:

reuseaddr

Allow servers to reuse a port without the system being completely sure the port is no longer in use.

bindtodevice(+*Device*)

Bind the socket to *Device* (an atom). For example, the code below binds the socket to the *loopback* device that is typically used to realise the *localhost*. See the manual pages for `setsockopt()` and the socket interface (e.g., `socket(7)` on Linux) for details.

```
tcp_socket(Socket),
tcp_setopt(Socket, bindtodevice(lo))
```

nodelay

nodelay(*true*)

If `true`, disable the Nagle optimization on this socket, which is enabled by default on almost all modern TCP/IP stacks. The Nagle optimization joins small packages, which is generally desirable, but sometimes not. Please note that the underlying TCP_NODELAY setting to `setsockopt()` is not available on all platforms and systems may require additional privileges to change this option. If the option is not supported, `tcp_setopt/2` raises a domain_error exception. See Wikipedia for details.

broadcast

UDP sockets only: broadcast the package to all addresses matching the address. The address is normally the address of the local subnet (i.e. 192.168.1.255). See `udp_send/4`.

ip_add_membership(+*MultiCastGroup*)

ip_add_membership(+*MultiCastGroup, +LocalInterface*)

ip_add_membership(+*MultiCastGroup, +LocalInterface, +InterfaceIndex*)

ip_drop_membership(+*MultiCastGroup*)

ip_drop_membership(+*MultiCastGroup, +LocalInterface*)

ip_drop_membership(+*MultiCastGroup, +LocalInterface, +InterfaceIndex*)

Join/leave a multicast group. Calls `setsockopt()` with the corresponding arguments.

dispatch(+*Boolean*)

In GUI environments (using XPCE or the Windows `swipl-win.exe` executable) this flags defines whether or not any events are dispatched on behalf of the user interface. Default is `true`. Only very specific situations require setting this to `false`.

tcp_fcntl(+*Stream, +Action, ?Argument*) *[det]*

Interface to the `fcntl()` call. Currently only suitable to deal switch stream to non-blocking mode using:

```
tcp_fcntl(Stream, setfl, nonblock),
```

An attempt to read from a non-blocking stream while there is no data available returns -1 (or `end_of_file` for `read/1`), but `at_end_of_stream/1` fails. On actual end-of-input, `at_end_of_stream/1` succeeds.

tcp_getopt(+*Socket, ?Option*) *[semidet]*

Get information about *Socket*. Defined properties are below. Requesting an unknown option results in a `domain_error` exception.

file_no(-*File*)

Get the OS file handle as an integer. This may be used for debugging and integration.

tcp_host_to_address(*?HostName, ?Address*) *[det]*

Translate between a machines host-name and it's (IP-)address. If *HostName* is an atom, it is resolved using `getaddrinfo()` and the IP-number is unified to *Address* using a term of the format `ip(Byte1,Byte2,Byte3,Byte4)`. Otherwise, if *Address* is bound to an `ip(Byte1,Byte2,Byte3,Byte4)` term, it is resolved by `gethostbyaddr()` and the canonical hostname is unified with *HostName*.

To be done This function should support more functionality provided by gethostbyaddr, probably by adding an option-list.

gethostname(-*Hostname*) *[det]*

Return the canonical fully qualified name of this host. This is achieved by calling `gethostname()` and return the canonical name returned by `getaddrinfo()`.

negotiate_socks_connection(+*DesiredEndpoint, +StreamPair*) *[det]*

Negotiate a connection to *DesiredEndpoint* over *StreamPair*. *DesiredEndpoint* should be in the form of either:

- hostname : port
- `ip(A,B,C,D)` : port

Errors `socks_error(Details)` if the SOCKS negotiation failed.

6.4 UDP protocol support

The current library provides limited support for UDP packets. The UDP protocol is a *connection-less* and *unreliable* datagram based protocol. That means that messages sent may or may not arrive at the client side and may arrive in a different order as they are sent. UDP messages are often used for streaming media or for service discovery using the broadcasting mechanism.

udp_socket(-*Socket*)

Similar to `tcp_socket/1`, but create a socket using the `SOCK_DGRAM` protocol, ready for UDP connections.

udp_receive(+*Socket, -Data, -From, +Options*)

Wait for and return the next datagram. The data is returned as a Prolog string object (see `string_to_list/2`). *From* is a term of the format ip(*A*,*B*,*C*,*D*):*Port* indicating the sender of the message. *Socket* can be waited for using `wait_for_input/3`. Defined *Options*:

as(+*Type*)

Defines the returned term-type. *Type* is one of `atom`, `codes` or `string` (default).

max_message_size(+*Size*)

Specify the maximum number of bytes to read from a UDP datagram. Size must be within the range 0-65535. If unspecified, a maximum of 4096 bytes will be read.

The typical sequence to receive UDP data is:

```
receive(Port) :-
        udp_socket(S),
        tcp_bind(S, Port),
        repeat,
            udp_receive(Socket, Data, From, [as(atom)]),
            format('Got ~q from ~q~n', [Data, From]),
            fail.
```

udp_send(+*Socket, +Data, +To, +Options*)

Send a UDP message. Data is a string, atom or code-list providing the data. *To* is an address of the form *Host*:*Port* where Host is either the hostname or a term `ip/4`. *Options* is currently unused.

A simple example to send UDP data is:

```
send(Host, Port, Message) :-
        udp_socket(S),
        udp_send(S, Message, Host:Port, []),
        tcp_close_socket(S).
```

A broadcast is achieved by using `tcp_setopt`(*Socket, broadcast*) prior to sending the datagram and using the local network broadcast address as a `ip/4` term.

The normal mechanism to discover a service on the local network is for the client to send a broadcast message to an agreed port. The server receives this message and replies to the client with a message indicating further details to establish the communication.

7 The stream_pool library

The `streampool` library dispatches input from multiple streams based on `wait_for_input/3`. It is part of the clib package as it is used most of the time together with the `socket` library. On non-Unix systems it often can only be used with socket streams.

With SWI-Prolog 5.1.x, multi-threading often provides a good alternative to using this library. In this schema one thread watches the listening socket waiting for connections and either creates a thread per connection or processes the accepted connections with a pool of *worker threads*. The library `http/thread_httpd` provides an example realising a mult-threaded HTTP server.

add_stream_to_pool(+*Stream, :Goal*)
: Add *Stream*, which must be an input stream and —on non-unix systems— connected to a socket to the pool. If input is available on *Stream*, *Goal* is called.

delete_stream_from_pool(+*Stream*)
: Delete the given stream from the pool. Succeeds, even if *Stream* is no member of the pool. If *Stream* is unbound the entire pool is emtied but unlike `close_stream_pool/0` the streams are not closed.

close_stream_pool
: Empty the pool, closing all streams that are part of it.

dispatch_stream_pool(+*TimeOut*)
: Wait for maximum of *TimeOut* for input on any of the streams in the pool. If there is input, call the *Goal* associated with `add_stream_to_pool/2`. If *Goal* fails or raises an exception a message is printed. *TimeOut* is described with `wait_for_input/3`.

  If *Goal* is called, there is *some* input on the associated stream. *Goal* must be careful not to block as this will block the entire pool.[1]

stream_pool_main_loop
: Calls `dispatch_stream_pool/1` in a loop until the pool is empty.

Below is a very simple example that reads the first line of input and echos it back.

```
:- use_module(library(streampool)).

server(Port) :-
        tcp_socket(Socket),
        tcp_bind(Socket, Port),
        tcp_listen(Socket, 5),
        tcp_open_socket(Socket, In, _Out),
        add_stream_to_pool(In, accept(Socket)),
        stream_pool_main_loop.

accept(Socket) :-
        tcp_accept(Socket, Slave, Peer),
        tcp_open_socket(Slave, In, Out),
        add_stream_to_pool(In, client(In, Out, Peer)).

client(In, Out, _Peer) :-
        read_line_to_codes(In, Command),
        close(In),
```

[1] This is hard to achieve at the moment as none of the Prolog read-commands provide for a timeout.

```
        format(Out, 'Please to meet you: ~s~n', [Command]),
        close(Out),
        delete_stream_from_pool(In).
```

8 library(uri): Process URIs

This library provides high-performance C-based primitives for manipulating URIs. We decided for a C-based implementation for the much better performance on raw character manipulation. Notably, URI handling primitives are used in time-critical parts of RDF processing. This implementation is based on RFC-3986:

```
http://labs.apache.org/webarch/uri/rfc/rfc3986.html
```

The URI processing in this library is rather liberal. That is, we break URIs according to the rules, but we do not validate that the components are valid. Also, percent-decoding for IRIs is liberal. It first tries UTF-8; then ISO-Latin-1 and finally accepts %-characters verbatim.

Earlier experience has shown that strict enforcement of the URI syntax results in many errors that are accepted by many other web-document processing tools.

uri_components(*+URI, -Components*) *[det]*
uri_components(*-URI, +Components*) *[det]*

Break a *URI* into its 5 basic components according to the RFC-3986 regular expression:

```
^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?
 12             3  4          5       6  7        8 9
```

Arguments

Components	is a term `uri_components(Scheme, Authority, Path, Search, Fragment)`. If a *URI* is **parsed**, i.e., using mode (+,-), components that are not found are left *uninstantiated* (variable). See `uri_data/3` for accessing this structure.

uri_data(*?Field, +Components, ?Data*) *[semidet]*

Provide access the uri_component structure. Defined field-names are: `scheme`, `authority`, `path`, `search` and `fragment`

uri_data(*+Field, +Components, +Data, -NewComponents*) *[semidet]*

NewComponents is the same as *Components* with *Field* set to *Data*.

uri_normalized(*+URI, -NormalizedURI*) *[det]*

NormalizedURI is the normalized form of *URI*. Normalization is syntactic and involves the following steps:

- 6.2.2.1. Case Normalization

- 6.2.2.2. Percent-Encoding Normalization
- 6.2.2.3. Path Segment Normalization

iri_normalized(+*IRI*, -*NormalizedIRI*) *[det]*

NormalizedIRI is the normalized form of *IRI*. Normalization is syntactic and involves the following steps:

- 6.2.2.1. Case Normalization
- 6.2.2.3. Path Segment Normalization

See also This is similar to `uri_normalized/2`, but does not do normalization of %-escapes.

uri_normalized_iri(+*URI*, -*NormalizedIRI*) *[det]*

As `uri_normalized/2`, but percent-encoding is translated into IRI Unicode characters. The translation is liberal: valid UTF-8 sequences of %-encoded bytes are mapped to the Unicode character. Other %XX-sequences are mapped to the corresponding ISO-Latin-1 character and sole % characters are left untouched.

See also `uri_iri/2`.

uri_is_global(+*URI*) *[semidet]*

True if *URI* has a scheme. The semantics is the same as the code below, but the implementation is more efficient as it does not need to parse the other components, nor needs to bind the scheme. The condition to demand a scheme of more than one character is added to avoid confusion with DOS path names.

```
uri_is_global(URI) :-
        uri_components(URI, Components),
        uri_data(scheme, Components, Scheme),
        nonvar(Scheme),
        atom_length(Scheme, Len),
        Len > 1.
```

uri_resolve(+*URI*, +*Base*, -*GlobalURI*) *[det]*

Resolve a possibly local *URI* relative to *Base*. This implements http://labs.apache.org/webarch/uri/rfc/rfc3986.html#relative-transform

uri_normalized(+*URI*, +*Base*, -*NormalizedGlobalURI*) *[det]*

NormalizedGlobalURI is the normalized global version of *URI*. Behaves as if defined by:

```
uri_normalized(URI, Base, NormalizedGlobalURI) :-
        uri_resolve(URI, Base, GlobalURI),
        uri_normalized(GlobalURI, NormalizedGlobalURI).
```

iri_normalized(+*IRI*, +*Base*, -*NormalizedGlobalIRI*) *[det]*

NormalizedGlobalIRI is the normalized global version of *IRI*. This is similar to `uri_normalized/3`, but does not do %-escape normalization.

uri_normalized_iri(+*URI, +Base, -NormalizedGlobalIRI*) *[det]*

NormalizedGlobalIRI is the normalized global IRI of *URI*. Behaves as if defined by:

```
uri_normalized(URI, Base, NormalizedGlobalIRI) :-
        uri_resolve(URI, Base, GlobalURI),
        uri_normalized_iri(GlobalURI, NormalizedGlobalIRI).
```

uri_query_components(+*String, -Query*) *[det]*

uri_query_components(-*String, +Query*) *[det]*

Perform encoding and decoding of an URI query string. *Query* is a list of fully decoded (Unicode) Name=Value pairs. In mode (-,+), query elements of the forms Name(Value) and Name-Value are also accepted to enhance interoperability with the option and pairs libraries. E.g.

```
?- uri_query_components(QS, [a=b, c('d+w'), n-'VU Amsterdam']).
QS = 'a=b&c=d%2Bw&n=VU%20Amsterdam'.

?- uri_query_components('a=b&c=d%2Bw&n=VU%20Amsterdam', Q).
Q = [a=b, c='d+w', n='VU Amsterdam'].
```

uri_authority_components(+*Authority, -Components*) *[det]*

uri_authority_components(-*Authority, +Components*) *[det]*

Break-down the authority component of a URI. The fields of the structure *Components* can be accessed using `uri_authority_data/3`.

uri_authority_data(+*Field, ?Components, ?Data*) *[semidet]*

Provide access the uri_authority structure. Defined field-names are: `user`, `password`, `host` and `port`

uri_encoded(+*Component, +Value, -Encoded*) *[det]*

uri_encoded(+*Component, -Value, +Encoded*) *[det]*

Encoded is the URI encoding for *Value*. When encoding (*Value->Encoded*), *Component* specifies the URI component where the value is used. It is one of `query_value`, `fragment`, `path` or `segment`. Besides alphanumerical characters, the following characters are passed verbatim (the set is split in logical groups according to RFC3986).

query_value, fragment "-._~" | "!$'()*,;" | "@" | "/?"

path "-._~" | "!$&'()*,;=" | "@" | "/"

segment "-._~" | "!$&'()*,;=" | "@"

uri_iri(+*URI, -IRI*) *[det]*

uri_iri(-*URI, +IRI*) *[det]*

Convert between a *URI*, encoded in US-ASCII and an *IRI*. An *IRI* is a fully expanded Unicode string. Unicode strings are first encoded into UTF-8, after which %-encoding takes place.

Errors `syntax_error(Culprit)` in mode (+,-) if *URI* is not a legally percent-encoded UTF-8 string.

uri_file_name(+*URI, -FileName*) *[semidet]*
uri_file_name(-*URI, +FileName*) *[det]*

Convert between a *URI* and a local file_name. This protocol is covered by RFC 1738. Please note that file-URIs use *absolute* paths. The mode (-, +) translates a possible relative path into an absolute one.

9 CGI Support library

This is currently a very simple library, providing support for obtaining the form-data for a CGI script:

cgi_get_form(-*Form*)

Decodes standard input and the environment variables to obtain a list of arguments passed to the CGI script. This predicate both deals with the CGI **GET** method as well as the **POST** method. If the data cannot be obtained, an `existence_error` exception is raised.

Below is a very simple CGI script that prints the passed parameters. To test it, compile this program using the command below, copy it to your cgi-bin directory (or make it otherwise known as a CGI-script) and make the query `http://myhost.mydomain/cgi-bin/cgidemo?hello=world`

```
% pl -o cgidemo --goal=main --toplevel=halt -c cgidemo.pl
```

```
:- use_module(library(cgi)).

main :-
        set_stream(current_output, encoding(utf8)),
        cgi_get_form(Arguments),
        format('Content-type: text/html; charset=UTF-8~n~n', []),
        format('<html>~n', []),
        format('<head>~n', []),
        format('<title>Simple SWI-Prolog CGI script</title>~n', []),
        format('</head>~n~n', []),
        format('<body>~n', []),
        format('<p>', []),
        print_args(Arguments),
        format('</body>~n</html>~n', []).

print_args([]).
print_args([A0|T]) :-
        A0 =.. [Name, Value],
        format('<b>~w</b>=<em>~w</em><br>~n', [Name, Value]),
        print_args(T).
```

9.1 Some considerations

Printing an HTML document using `format/2` is not a neat way of producing HTML because it is vulnerable to required escape sequences. A high-level alternative is provided by `http/html_write` from the HTTP library.

The startup-time of Prolog is relatively long, in particular if the program is large. In many cases it is much better to use the SWI-Prolog HTTP server library and make the main web-server relay requests to the SWI-Prolog webserver. See the SWI-Prolog HTTP package for details.

The CGI standard is unclear about handling Unicode data. The above two declarations ensure the CGI script will send all data in UTF-8 and thus provide full support of Unicode. It is assumed that browsers generally send form-data using the same encoding as the page in which the form appears, UTF-8 or ISO Latin-1. The current version of `cgi_get_form/1` assumes the CGI data is in UTF-8.

10 Password encryption library

The `crypt` library defines `crypt/2` for encrypting and testing passwords. The clib package also provides crytographic hashes as described in section 12

crypt(+*Plain, ?Encrypted*)

This predicate can be used in three modes. To test whether a password matches an encrypted version thereof, simply run with both arguments fully instantiated. To generate a default encrypted version of *Plain*, run with unbound *Encrypted* and this argument is unified to a list of character codes holding an encrypted version.

The library supports two encryption formats: traditional Unix DES-hashes[2] and FreeBSD compatible MD5 hashes (all platforms). MD5 hashes start with the magic sequence `$1$`, followed by an up to 8 character *salt*. DES hashes start with a 2 character *salt*. Note that a DES hash considers only the first 8 characters. The MD5 considers the whole string.

Salt and algorithm can be forced by instantiating the start of *Encrypted* with it. This is typically used to force MD5 hashes:

```
?- phrase("$1$", E, _),
   crypt("My password", E),
   format('~s~n', [E]).

$1$qdaDeDZn$ZUxSQEESEHIDCHPNc3fxZ1
```

Encrypted is always a list of ASCII character codes. *Plain* only supports ISO-Latin-1 passwords in the current implementation.

Plain is either an atom, SWI-Prolog string, list of characters or list of character-codes. It is not advised to use atoms, as this implies the password will be available from the Prolog heap as a defined atom.

NOTE: `crypt/2` provides an interface to the Unix password hashing API. Above we already introduced support for classical DES and MD5 hashes, both hashes that are considered *insecure*

[2] On non-Unix systems, crypt() is provided by the NetBSD library. The license header is added at the end of this document.

by today's standards.[3] The crypt() API of modern Unix systems typically support more secure hashes. Using `crypt/2` is suitable if compatibility with OS passwords is required. If strong hashes and platform independence are important to you, use `crypto_password_hash/2` provided by library `crypto` from the ssl package.

11 library(uuid): Universally Unique Identifier (UUID) Library

See also http://www.ossp.org/pkg/lib/uuid/

To be done Compare UUIDs, extract time and version from UUIDs

The library provides operations on UUIDs. Please consult other sources for understanding UUIDs and the implications of the different UUID versions. Some typical calls are given below:

```
?- uuid(X).
X = 'ea6589fa-19dd-11e2-8a49-001d92e1879d'.

?- uuid(X, [url('http://www.swi-prolog.org')]).
X = '73a07870-6a90-3f2e-ae2b-ffa538dc7c2c'.
```

uuid(-*UUID*) *[det]*

UUID is an atom representing a new *UUID*. This is the same as calling `uuid(UUID, [])`. See `uuid/2` for options.

uuid(-*UUID*, +*Options*) *[det]*

Create a new *UUID* according to *Options*. The following options are defined:

version(+*Versions*)

Integer in the range 1..5, which specifies the *UUID* version that is created. Default is 1.

dns(*DNS*)

url(*URL*)

oid(*OID*)

x500(*X500*)

Provide additional context information for UUIDs using version 3 or 5. If there is no explicit version option, *UUID* version 3 is used.

format(+*Format*)

Representation of the *UUID*. Default is `atom`, yielding atoms such as `8304efdd-bd6e-5b7c-a27f-83f3f05c64e0`. The alternative is `integer`, returning a large integer that represents the 128 bits of the *UUID*.

[3] *Insecure* means that the password can realistically be derived from the password hash using a brute-force attack. This implies that leaking the password database is an immediate security risk.

12 SHA* Secure Hash Algorithms

The library `sha` provides *Secure Hash Algorihms* approved by FIPS (*Federal Information Processing Standard*). Quoting Wikipedia: *"The SHA (Secure Hash Algorithm) hash functions refer to five FIPS-approved algorithms for computing a condensed digital representation (known as a message digest) that is, to a high degree of probability, unique for a given input data sequence (the message). These algorithms are called 'secure' because (in the words of the standard), "for a given algorithm, it is computationally infeasible 1) to find a message that corresponds to a given message digest, or 2) to find two different messages that produce the same message digest. Any change to a message will, with a very high probability, result in a different message digest."*

The current library supports all 5 approved algorithms, both computing the hash-key from data and the *hash Message Authentication Code* (HMAC).

A general secure hash interface is provided by `crypto`, part of the ssl package.

Input is text, represented as an atom, packed string object or code-list. Note that these functions operate on byte-sequences and therefore are not meaningful on Unicode text. The result is returned as a list of byte-values. This is the most general format that is comfortable supported by standard Prolog and can easily be transformed in other formats. Commonly used text formats are ASCII created by encoding each byte as two hexadecimal digits and ASCII created using *base64* encoding. Representation as a large integer can be desirable for computational processing.

sha_hash(+*Data, -Hash, +Options*)

Hash is the SHA hash of Data. *Data* is either an atom, packed string or list of character codes. *Hash* is unified with a list of bytes (integers in the range 0..255) representing the hash. See `hash_atom/2` to convert this into the more commonly seen hexadecimal representation. The conversion is controlled by Options:

algorithm(+*Algorithm*)

One of `sha1` (default), `sha224`, `sha256`, `sha384` or `sha512`

encoding(+*Encoding*)

This option defines the mapping from Prolog (Unicode) text to bytes on which the SHA algorithm is performed. It has two values. The defualt is `utf8`, which implies that Unicode text is encoded as UTF-8 bytes. This option can deal with any atom. The alternative is `octet`, which implies that the text is considered as a sequence of bytes. This is suitable for e.g., atoms that represent binary data. An error is raised if the text contains code-points outside the range 0..255.

hmac_sha(+*Key, +Data, -HMAC, +Options*)

Quoting Wikipedia: *"A keyed-hash message authentication code, or HMAC, is a type of message authentication code (MAC) calculated using a cryptographic hash function in combination with a secret key. As with any MAC, it may be used to simultaneously verify both the data integrity and the authenticity of a message. Any iterative cryptographic hash function, such as MD5 or SHA-1, may be used in the calculation of an HMAC; the resulting MAC algorithm is termed HMAC-MD5 or HMAC-SHA-1 accordingly. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, on the size and quality of the key and the size of the hash output length in bits."*

Key and *Data* are either an atom, packed string or list of character codes. *HMAC* is unified with a list of integers representing the authentication code. *Options* is the same as for `sha_hash/3`,

but currently only `sha1` and `sha256` are supported.

hash_atom(+*Hash, -HexAtom*)
True when *HexAtom* is the commonly used hexadecimal encoding of the hash code. E.g.,

```
?- sha_hash('SWI-Prolog', Hash, []),
   hash_atom(Hash, Hex).
Hash = [61, 128, 252, 38, 121, 69, 229, 85, 199|...],
Hex = '3d80fc267945e555c730403bd0ab0716e2a68c68'.
```

12.1 License terms

The underlying SHA-2 library is an unmodified copy created by Dr Brian Gladman, Worcester, UK. It is distributed under the license conditions below.

The free distribution and use of this software in both source and binary form is allowed (with or without changes) provided that:

1. distributions of this source code include the above copyright notice, this list of conditions and the following disclaimer;
2. distributions in binary form include the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other associated materials;
3. the copyright holder's name is not used to endorse products built using this software without specific written permission.

ALTERNATIVELY, provided that this notice is retained in full, this product may be distributed under the terms of the GNU General Public License (GPL), in which case the provisions of the GPL apply INSTEAD OF those given above.

13 library(md5): MD5 hashes

See also `library(sha)`, `library(hash_stream)` and `library(crypto)`.

Compute MD5 hashes from a Prolog string. This library provides a lightweight alternative to the general secure hash interface provided by `library(crypto)` from the `ssl` package.

md5_hash(+*Data, -Hash, +Options*) *[det]*
Hash is the MD5 hash of *Data*, The conversion is controlled by *Options*:

encoding(+*Encoding*)
If *Data* is a sequence of character *codes*, this must be translated into a sequence of *bytes*, because that is what the hashing requires. The default encoding is `utf8`. The other meaningful value is `octet`, claiming that *Data* contains raw bytes.

Arguments

Data	is either an atom, string, code-list or char-list.
Hash	is an atom holding 32 characters, representing the hash in hexadecimal notation

14 library(hash_stream): Maintain a hash on a stream

See also In addition to this hash library, SWI-Prolog provides `library(md5)`, `library(sha)` and hash functions through `library(crypto)`, part of the `ssl` package.

This library defines a filter stream that maintains a hash of the data that passes through the stream. It can be used to compute the hash of input data while it is being processed. This is notably interesting if data is processed from a socket as it avoids the need for collecting the data first in a temporary file.

A typical processing sequence is illustrated below, where `process/2` somehow processed the data and `save_result/3` records the result as obtained from *URL* with content digest *SHA256* its *Result*.

```
    ...,
    http_open(URL, In0, []),
    open_hash_stream(In0, In, [algorithm(sha256)]),
    process(In, Result),
    stream_hash(In, SHA256),
    close(In),
    save_result(URL, SHA256, Result)
```

This library can also be used to compute the hash for the content of a file. The advantage is that this code doesn't rely on external tools. It is considerably faster for short files, but considerably slower on large files because Prolog I/O is based on character streams rather than blocks.

```
file_hash(Algorithm, File, Hash) :-
    setup_call_cleanup(
        open(File, read, In0, [type(binary)]),
        setup_call_cleanup(
            open_hash_stream(In0, In,
                             [ algorithm(Algorithm),
                               close_parent(false)
                             ]),
            ( setup_call_cleanup(
                  open_null_stream(Null),
                  copy_stream_data(In, Null),
                  close(Null)),
              stream_hash(In, Hash)
            ),
            close(In)),
        close(In0)).
```

open_hash_stream(+*OrgStream, -HashStream, +Options*) *[det]*

Open a filter stream on *OrgStream* that maintains a hash. The hash can be retrieved at any time using `stream_hash/2`. Provided options:

algorithm(+*Algorithm*)

One of `md5`, `sha1`, `sha224`, `sha256`, `sha384` or `sha512`. Default is `sha1`.

close_parent(+*Bool*)

If `true` (default), closing the filter stream also closes the original (parent) stream.

stream_hash(+*HashStream, -Digest:atom*) *[det]*

Unify *Digest* with a hash for the bytes send to or read from *HashStream*. Note that the hash is computed on the stream buffers. If the stream is an output stream, it is first flushed and the *Digest* represents the hash at the current location. If the stream is an input stream the *Digest* represents the hash of the processed input including the already buffered data.

15 Memory files

The `memfile` provides an alternative to temporary files, intended for temporary buffering of data. Memory files in general are faster than temporary files and do not suffer from security risks or naming conflicts associated with temporary-file management.

There is no limit to the number of memory streams, nor the size of them. However, a single memory file cannot have multiple streams at the same time, i.e., a memory file cannot be opened multiple times, not even for reading. Memory files are thread-safe and subject to (atom) garbage collection.

These predicates are first of all intended for building higher-level primitives such as `open_codes_stream/3`. See also `format/3`, `atom_to_term/3`, `term_to_atom/2`, `term_string/2`, etc.

new_memory_file(-*Handle*)

Create a new memory file and return a unique opaque handle to it.

free_memory_file(+*Handle*)

Discard the memory file and its contents. If the file is open it is first closed.

open_memory_file(+*Handle, +Mode, -Stream*)

Open the memory-file. *Mode* is one of `read`, `write`, `append`, `update` or `insert`. The resulting *Stream* must be closed using `close/1`. When opened for `update` or `insert`, the current location is initialized at the start of the data and can be modified using `seek/2` or `set_stream_position/2`. In `update` mode, existing content is replaced, while the size is enlarged after hitting the end of the data. In `insert` mode, the new data is inserted at the current point.

open_memory_file(+*Handle, +Mode, -Stream, +Options*)

Open a memory-file as `open_memory_file/3`. Options:

encoding(+*Encoding*)

Set the encoding for a memory file and the created stream. Encoding names are the same as used with `open/4`. By default, memoryfiles represent UTF-8 streams, making them capable of storing arbitrary Unicode text. In practice the only alternative is `octet`, turning the memoryfile into binary mode. Please study SWI-Prolog Unicode and encoding issues before using this option.

free_on_close(+*Bool*)

If `true` (default `false` and the memory file is opened for reading, discard the file (see `free_memory_file/1`) if the input is closed. This is used to realise `open_chars_stream/2` in library(charsio).

size_memory_file(+*Handle, -Size*)

Return the content-length of the memory-file in characters in the current encoding of the memory file. The file should be closed and contain data.

size_memory_file(+*Handle, -Size, +Encoding*)

Return the content-length of the memory-file in characters in the given *Encoding*. The file should be closed and contain data.

atom_to_memory_file(+*Atom, -Handle*)

Turn an atom into a read-only memory-file containing the (shared) characters of the atom. Opening this memory-file in mode `write` yields a permission error.

insert_memory_file(+*Handle, +Offset, +Data*)

Insert *Data* into the memory file at location *Offset*. The offset is specified in characters. *Data* can be an atom, string, code or character list. Other terms are first serialized using `writeq/1`. This predicate raises a domain_error exception if *Offset* is out of range and a permission_error if the memory file is read-only or opened.

delete_memory_file(+*Handle, +Offset, +Length*)

Delete a *Length* characters from the memory file, starting at *Offset*. This predicate raises a domain_error exception if *Offset* or *Offset+Length* is out of range and a permission_error if the memory file is read-only or opened.

memory_file_to_atom(+*Handle, -Atom*)

Return the content of the memory-file in *Atom*.

memory_file_to_atom(+*Handle, -Atom, +Encoding*)

Return the content of the memory-file in *Atom*, pretending the data is in the given *Encoding*. This can be used to convert from one encoding into another, typically from/to bytes. For example, if we must convert a set of bytes that contain text in UTF-8, open the memory file as octet stream, fill it, and get the result using *Encoding* is `utf8`.

memory_file_to_codes(+*Handle, -Codes*)

Return the content of the memory-file as a list of character-codes in *Codes*.

memory_file_to_codes(+*Handle, -Codes, +Encoding*)

Return the content of the memory-file as a list of character-codes in *Codes*, pretending the data is in the given *Encoding*.

memory_file_to_string(+*Handle, -String*)

Return the content of the memory-file as a string in *-String*.

memory_file_to_string(+*Handle, -String, +Encoding*)

Return the content of the memory-file as a string in *String*, pretending the data is in the given *Encoding*.

memory_file_substring(+*Handle, ?Before, ?Length, ?After, -SubString***)**

SubString is a substring of the memory file. There are *Before* characters in the memory file before *SubString*, *SubString* contains *Length* character and is followed by *After* characters in the memory file. The signature is the same as `sub_string/5` and `sub_atom/5`, but currently at least two of the 3 position arguments must be specified. Future versions might implement the full functionality of `sub_string/5`.

memory_file_line_position(+*MF, ?Line, ?LinePos, ?Offset***)**

True if the character offset *Offset* corresponds with the *LinePos* character on line *Line*. Lines are counted from one (1). Note that *LinePos* is *not* the *column* as each character counts for one, including backspace and tab.

16 Time and alarm library

The `time` provides timing and alarm functions. Alarms are thread-specific, i.e., creating an alarm causes the alarm goal to be called in the thread that created it. The predicate `current_alarm/4` only reports alarms that are related to the calling thread. If a thread terminates, all remaining alarms are silently removed. Most applications use `call_with_time_limit/2`.

alarm(+*Time, :Callable, -Id, +Options***)**

Schedule *Callable* to be called *Time* seconds from now. *Time* is a number (integer or float). *Callable* is called on the next pass through a call- or redo-port of the Prolog engine, or a call to the `PL_handle_signals()` routine from SWI-Prolog. *Id* is unified with a reference to the timer.

The resolution of the alarm depends on the underlying implementation, which is based on pthread_cond_timedwait() (on Windows on the pthread emulation thereof). Long-running foreign predicates that do not call `PL_handle_signals()` may further delay the alarm. The relation to blocking system calls (sleep, reading from slow devices, etc.) is undefined and varies between implementations.

Options is a list of *Name*(*Value*) terms. Defined options are:

remove(*Bool***)**

If `true` (default `false`), the timer is removed automatically after fireing. Otherwise it must be destroyed explicitly using `remove_alarm/1`.

install(*Bool***)**

If `false` (default `true`), the timer is allocated but not scheduled for execution. It must be started later using `install_alarm/1`.

alarm(+*Time, :Callable, -Id***)**

Same as `alarm`(*Time, Callable, Id, []*).

alarm_at(+*Time, :Callable, -Id, +Options***)**

as `alarm/3`, but *Time* is the specification of an absolute point in time. Absolute times are specified in seconds after the Jan 1, 1970 epoch. See also `date_time_stamp/2`.

install_alarm(+*Id***)**

Activate an alarm allocated using `alarm/4` with the option `install`(*false*) or stopped using `uninstall_alarm/1`.

install_alarm(+*Id, +Time*)

As `install_alarm/1`, but specifies a new (relative) timeout value.

uninstall_alarm(+*Id*)

Deactivate a running alarm, but do not invalidate the alarm identifier. Later, the alarm can be reactivated using either `install_alarm/1` or `install_alarm/2`. Reinstalled using `install_alarm/1`, it will fire at the originally scheduled time. Reinstalled using `install_alarm/2` causes the alarm to fire at the specified time from now.

remove_alarm(+*Id*)

Remove an alarm. If it is not yet fired, it will not be fired any more.

current_alarm(*?At, ?:Callable, ?Id, ?Status*)

Enumerate the not-yet-removed alarms. *Status* is one of `done` if the alarm has been called, `next` if it is the next to be fired and *scheduled* otherwise.

call_with_time_limit(+*Time, :Goal*)

True if *Goal* completes within *Time* seconds. *Goal* is executed as in `once/1`. If *Goal* doesn't complete within *Time* seconds (wall time), exit using the exception `time_limit_exceeded`. See `catch/3`.

Please note that this predicate uses `alarm/4` and therefore its effect on long-running foreign code and system calls is undefined. Blocking I/O can be handled using the timeout option of `read_term/3`.

17 library(unix): Unix specific operations

See also `library(process)` provides a portable high level interface to create and manage processes.

The `library(unix)` library provides the commonly used Unix primitives to deal with process management. These primitives are useful for many tasks, including server management, parallel computation, exploiting and controlling other processes, etc.

The predicates in this library are modelled closely after their native Unix counterparts.

fork(-*Pid*) *[det]*

Clone the current process into two branches. In the child, *Pid* is unified to child. In the original process, *Pid* is unified to the process identifier of the created child. Both parent and child are fully functional Prolog processes running the same program. The processes share open I/O streams that refer to Unix native streams, such as files, sockets and pipes. Data is not shared, though on most Unix systems data is initially shared and duplicated only if one of the programs attempts to modify the data.

Unix `fork()` is the only way to create new processes and `fork/1` is a simple direct interface to it.

Errors `permission_error(fork, process, main)` is raised if the calling thread is not the only thread in the process. Forking a Prolog process with threads will typically deadlock because only the calling thread is cloned in the fork, while all thread synchronization are cloned.

fork_exec(+*Command*) *[det]*

Fork (as `fork/1`) and exec (using `exec/1`) the child immediately. This behaves as the code below, but bypasses the check for the existence of other threads because this is a safe scenario.

```
fork_exec(Command) :-
      (   fork(child)
      ->  exec(Command)
      ;   true
      ).
```

exec(+*Command*)

Replace the running program by starting *Command*. *Command* is a callable term. The functor is the command and the arguments provide the command-line arguments for the command. Each command-line argument must be atomic and is converted to a string before passed to the Unix call `execvp()`. Here are some examples:

- `exec(ls('-l'))`
- `exec('/bin/ls'('-l', '/home/jan'))`

Unix `exec()` is the only way to start an executable file executing. It is commonly used together with `fork/1`. For example to start netscape on an URL in the background, do:

```
run_netscape(URL) :-
        (    fork(child),
             exec(netscape(URL))
        ;    true
        ).
```

Using this code, netscape remains part of the process-group of the invoking Prolog process and Prolog does not wait for netscape to terminate. The predicate `wait/2` allows waiting for a child, while `detach_IO/0` disconnects the child as a deamon process.

wait(*?Pid, -Status*) *[det]*

Wait for a child to change status. Then report the child that changed status as well as the reason. If *Pid* is bound on entry then the status of the specified child is reported. If not, then the status of any child is reported. *Status* is unified with `exited(ExitCode)` if the child with pid *Pid* was terminated by calling `exit()` (Prolog `halt/1`). ExitCode is the return status. *Status* is unified with `signaled(Signal)` if the child died due to a software interrupt (see `kill/2`). Signal contains the signal number. Finally, if the process suspended execution due to a signal, *Status* is unified with `stopped(Signal)`.

kill(+*Pid, +Signal*) *[det]*

Deliver a software interrupt to the process with identifier *Pid* using software-interrupt number *Signal*. See also `on_signal/2`. Signals can be specified as an integer or signal name, where signal names are derived from the C constant by dropping the `SIG` prefix and mapping to lowercase. E.g. `int` is the same as `SIGINT` in C. The meaning of the signal numbers can be found in the Unix manual.

pipe(*-InSream, -OutStream*) *[det]*

Create a communication-pipe. This is normally used to make a child communicate to its parent. After `pipe/2`, the process is cloned and, depending on the desired direction, both processes close the end of the pipe they do not use. Then they use the remaining stream to communicate. Here is a simple example:

```
:- use_module(library(unix)).

fork_demo(Result) :-
        pipe(Read, Write),
        fork(Pid),
        (   Pid == child
        ->  close(Read),
            format(Write, '~q.~n',
                   [hello(world)]),
            flush_output(Write),
            halt
        ;   close(Write),
            read(Read, Result),
            close(Read)
        ).
```

dup(*+FromStream, +ToStream*) *[det]*

Interface to Unix `dup2()`, copying the underlying filedescriptor and thus making both streams point to the same underlying object. This is normally used together with `fork/1` and `pipe/2` to talk to an external program that is designed to communicate using standard I/O.

Both *FromStream* and *ToStream* either refer to a Prolog stream or an integer descriptor number to refer directly to OS descriptors. See also `demo/pipe.pl` in the source-distribution of this package.

detach_IO(*+Stream*) *[det]*

This predicate is intended to create Unix *deamon* processes. It performs two actions.

1. The I/O streams `user_input`, `user_output` and `user_error` are closed if they are connected to a terminal (see `tty` property in `stream_property/2`). Input streams are rebound to a dummy stream that returns EOF. Output streams are reboud to forward their output to *Stream*.
2. The process is detached from the current process-group and its controlling terminal. This is achieved using `setsid()` if provided or using `ioctl() TIOCNOTTY` on `/dev/tty`.

To ignore all output, it may be rebound to a null stream. For example:

```
    ...,
    open_null_stream(Out),
    detach_IO(Out).
```

The `detach_IO/1` should be called only once per process. Subsequent calls silently succeed without any side effects.

See also `detach_IO/0` and `library(syslog)`.

detach_IO *[det]*

Detach I/O similar to `detach_IO/1`. The output streams are bound to a file `/tmp/pl-out.<pid>`. Output is line buffered (see `set_stream/2`).

See also `library(syslog)` allows for sending output to the Unix logging service.
Compatibility Older versions of this predicate only created this file if there was output.

prctl(+*Option*) *[det]*

Access to Linux process control operations. Defines values for *Option* are:

set_dumpable(+*Boolean*)
Control whether the process is allowed to dump core. This right is dropped under several uid and gid conditions.

get_dumpable(-*Boolean*)
Get the value of the dumpable flag.

sysconf(+*Conf*) *[semidet]*

Access system configuration. See `sysconf(1)` for details. *Conf* is a term Config(Value), where Value is always an integer. Config is the `sysconf()` name after removing =_SC_= and conversion to lowercase. Currently support the following configuration info: `arg_max`, `child_max`, `clk_tck`, `open_max`, `pagesize`, `phys_pages`, `avphys_pages`, `nprocessors_conf` and `nprocessors_onln`. Note that not all values may be supported on all operating systems.

18 Limiting process resources

The `rlimit` library provides an interface to the POSIX getrlimit()/setrlimit() API that control the maximum resource-usage of a process or group of processes. This call is especially useful for servers such as CGI scripts and inetd-controlled servers to avoid an uncontrolled script claiming too much resources.

rlimit(+*Resource, -Old, +New*)

Query and/or set the limit for *Resource*. Time-values are in seconds and size-values are counted in bytes. The following values are supported by this library. Please note that not all resources may be available and accessible on all platforms. This predicate can throw a variety of exceptions. In portable code this should be guarded with `catch/3`. The defined resources are:

`as`	Max address space
`cpu`	CPU time in seconds
`fsize`	Maximum filesize
`data`	max data size
`stack`	max stack size
`core`	max core file size
`rss`	max resident set size
`nproc`	max number of processes
`nofile`	max number of open files
`memlock`	max locked-in-memory address

When the process hits a limit POSIX systems normally send the process a signal that terminates it. These signals may be caught using SWI-Prolog's `on_signal/3` primitive. The code below illustrates this behaviour. Please note that asynchronous signal handling is dangerous, especially when using threads. 100% fail-safe operation cannot be guaranteed, but this procedure will inform the user properly 'most of the time'.

```
rlimit_demo :-
        rlimit(cpu, _, 2),
        on_signal(xcpu, _, cpu_exceeded),
        ( repeat, fail ).

cpu_exceeded(_Sig) :-
        format(user_error, 'CPU time exceeded~n', []),
        halt(1).
```

19 library(udp_broadcast): A UDP broadcast proxy

author Jeffrey Rosenwald (JeffRose@acm.org), Jan Wielemaker
See also `tipc.pl`
license BSD-2

SWI-Prolog's broadcast library provides a means that may be used to facilitate publish and subscribe communication regimes between anonymous members of a community of interest. The members of the community are however, necessarily limited to a single instance of Prolog. The UDP broadcast library removes that restriction. With this library loaded, any member on your local IP subnetwork that also has this library loaded may hear and respond to your broadcasts.

This library support three styles of networking as described below. Each of these networks have their own advantages and disadvantages. Please study the literature to understand the consequences.

broadcast Broadcast messages are sent to the LAN subnet. The broadcast implementation uses two UDP ports: a public to address the whole group and a private one to address a specific node. Broadcasting is generally a good choice if the subnet is small and traffic is low.

unicast Unicast sends copies of packages to known peers. Unicast networks can easily be routed. The unicast version uses a single UDP port per node. Unicast is generally a good choice for a small party, in particular if the peers are in different networks.

multicast Multicast is like broadcast, but it can be configured to work accross networks and may work more efficiently on VLAN networks. Like the broadcast setup, two UDP ports are used. Multicasting can in general deliver the most efficient LAN and WAN networks, but requires properly configured routing between the peers.

After initialization and, in the case of a *unicast* network managing the set of peers, communication happens through `broadcast/1`, `broadcast_request/1` and `listen/1`,2,3.

A `broadcast/1` or `broadcast_request/1` of the shape `udp(Scope, Term)` or `udp(Scope, Term, TimeOut)` is forwarded over the UDP network to all peers that joined the same *Scope*. To prevent the potential for feedback loops, only the plain *Term* is broadcasted locally. The timeout is optional. It specifies the amount to time to wait for replies to arrive in response to a `broadcast_request/1`. The default period is 0.250 seconds. The timeout is ignored for broadcasts.

An example of three separate processes cooperating in the same *scope* called `peers`:

```
Process A:

   ?- listen(number(X), between(1, 5, X)).
   true.

   ?-

Process B:

   ?- listen(number(X), between(7, 9, X)).
   true.

   ?-

Process C:

   ?- findall(X, broadcast_request(udp(peers, number(X))), Xs).
   Xs = [1, 2, 3, 4, 5, 7, 8, 9].

   ?-
```

It is also possible to carry on a private dialog with a single responder. To do this, you supply a compound of the form, Term:PortId, to a UDP scoped `broadcast/1` or `broadcast_request/1`, where PortId is the ip-address and port-id of the intended listener. If you supply an unbound variable, PortId, to broadcast_request, it will be unified with the address of the listener that responds to Term. You may send a directed broadcast to a specific member by simply providing this address in a similarly structured compound to a UDP scoped `broadcast/1`. The message is sent via unicast to that member only by way of the member's broadcast listener. It is received by the listener just as any other broadcast would be. The listener does not know the difference.

For example, in order to discover who responded with a particular value:

```
Host B Process 1:
```

```
   ?- listen(number(X), between(1, 5, X)).
   true.

   ?-

Host A Process 1:


   ?- listen(number(X), between(7, 9, X)).
   true.

   ?-

Host A Process 2:

   ?- listen(number(X), between(1, 5, X)).
   true.

   ?- bagof(X, broadcast_request(udp(peers,number(X):From,1)), Xs).
   From = ip(192, 168, 1, 103):34855,
   Xs = [7, 8, 9] ;
   From = ip(192, 168, 1, 103):56331,
   Xs = [1, 2, 3, 4, 5] ;
   From = ip(192, 168, 1, 104):3217,
   Xs = [1, 2, 3, 4, 5].
```

All incomming trafic is handled by a single thread with the alias `udp_inbound_proxy`. This thread also performs the internal dispatching using `broadcast/1` and `broadcast_request/1`. Future versions may provide for handling these requests in seperate threads.

19.1 Caveats

While the implementation is mostly transparent, there are some important and subtle differences that must be taken into consideration:

- UDP broadcast requires an initialization step in order to launch the broadcast listener proxy. See `udp_broadcast_initialize/2`.
- Prolog's `broadcast_request/1` is nondet. It sends the request, then evaluates the replies synchronously, backtracking as needed until a satisfactory reply is received. The remaining potential replies are not evaluated. With UDP, all peers will send all answers to the query. The receiver may however stop listening.
- A UDP `broadcast/1` is completely asynchronous.
- A UDP `broadcast_request/1` is partially synchronous. A `broadcast_request/1` is sent, then the sender balks for a period of time (default: 250 ms) while the replies are collected. Any reply that is received after this period is silently discarded. A optional second argument is provided so that a sender may specify more (or less) time for replies.

- Replies are presented to the user as a choice point on arrival, until the broadcast request timer finally expires. This allows traffic to propagate through the system faster and provides the requestor with the opportunity to terminate a broadcast request early if desired, by simply cutting choice points.
- Please beware that broadcast request transactions remain active and resources consumed until broadcast_request finally fails on backtracking, an uncaught exception occurs, or until choice points are cut. Failure to properly manage this will likely result in chronic exhaustion of UDP sockets.
- If a listener is connected to a generator that always succeeds (e.g. a random number generator), then the broadcast request will never terminate and trouble is bound to ensue.
- `broadcast_request/1` with `udp_subnet` scope is *not* reentrant. If a listener performs a `broadcast_request/1` with UDP scope recursively, then disaster looms certain. This caveat does not apply to a UDP scoped `broadcast/1`, which can safely be performed from a listener context.
- UDP broadcast's capacity is not infinite. While it can tolerate substantial bursts of activity, it is designed for short bursts of small messages. Unlike TIPC, UDP is unreliable and has no QOS protections. Congestion is likely to cause trouble in the form of non-Byzantine failure. That is, late, lost (e.g. infinitely late), or duplicate datagrams. Caveat emptor.
- A UDP `broadcast_request/1` term that is grounded is considered to be a broadcast only. No replies are collected unless the there is at least one unbound variable to unify.
- A UDP `broadcast/1` always succeeds, even if there are no listeners.
- A UDP `broadcast_request/1` that receives no replies will fail.
- Replies may be coming from many different places in the network (or none at all). No ordering of replies is implied.
- Prolog terms are sent to others after first converting them to atoms using `term_string/3`. Serialization does not deal with cycles, attributes or sharing. The hook `udp_term_string_hook/3` may be defined to change the message serialization and support different message formats and/or encryption.
- The broadcast model is based on anonymity and a presumption of trust–a perfect recipe for compromise. UDP is an Internet protocol. A UDP broadcast listener exposes a public port, which is static and shared by all listeners, and a private port, which is semi-static and unique to the listener instance. Both can be seen from off-cluster nodes and networks. Usage of this module exposes the node and consequently, the cluster to significant security risks. So have a care when designing your application. You must talk only to those who share and contribute to your concerns using a carefully prescribed protocol.
- UDP broadcast categorically and silently ignores all message traffic originating from or terminating on nodes that are not members of the local subnet. This security measure only keeps honest people honest!

udp_broadcast_close(+*Scope*)
Close a UDP broadcast scope.

udp_broadcast_initialize(+*IPAddress, +Options*) *[semidet]*
Initialized UDP broadcast bridge. *IPAddress* is the IP address on the network we want to broadcast on. IP addresses are terms `ip(A,B,C,D)` or an atom or string of the format `A.B.C.D`. *Options* processed:

scope(+*ScopeName*)
Name of the scope. Default is `subnet`.

subnet_mask(+*SubNet*)
Subnet to broadcast on. This uses the same syntax as *IPAddress*. Default classifies the network as class A, B or C depending on the the first octet and applies the default mask.

port(+*Port*)
Public port to use. Default is 20005.

method(+*Method*)
Method to send a message to multiple peers. One of

broadcast
Use UDP broadcast messages to the LAN. This is the default

multicast
Use UDP multicast messages. This can be used on WAN networks, provided the intermediate routers understand multicast.

unicast
Send the messages individually to all registered peers.

For compatibility reasons *Options* may be the subnet mask.

udp_peer_add(+*Scope, +Address*) *[det]*
udp_peer_del(+*Scope, ?Address*) *[det]*
udp_peer(*?Scope, ?Address*) *[nondet]*
Manage and query the set of known peers for a unicast network. *Address* is either a term IP:Port or a plain IP address. In the latter case the default port registered with the scope is used.

Arguments

Address	has canonical form `ip(A,B,C,D)`:Port.

udp_unicast_join_hook(+*Scope, +From, +Data*) *[semidet,multifile]*
This multifile hook is called if an UDP package is received on the port of the unicast network identified by *Scope*. *From* is the origin IP and port and *Data* is the message data that is deserialized as defined for the scope (see `udp_term_string/3`).

This hook is intended to initiate a new node joining the network of peers. We could in theory also omit the in-scope test and use a normal broadcast to join. Using a different channal however provides a basic level of security. A possibe implementation is below. The first fragment is a hook added to the server, the second is a predicate added to a client and the last initiates the request in the client. The excanged term (`join(X)`) can be used to exchange a welcome handshake.

```
:- multifile udp_broadcast:udp_unicast_join_hook/3.
udp_broadcast:udp_unicast_join_hook(Scope, From, join(welcome)) :-
    udp_peer_add(Scope, From),
```

```
join_request(Scope, Address, Reply) :-
    udp_peer_add(Scope, Address),
    broadcast_request(udp(Scope, join(X))).
```

```
?- join_request(myscope, "1.2.3.4":10001, Reply).
Reply = welcome.
```

20 library(prolog_stream): A stream with Prolog callbacks

This library defines a Prolog stream that realises its low-level I/O with callbacks to Prolog. The library was developed to bind normal Prolog I/O to Pengines I/O. This type of I/O redirection is probably the primary use case.

open_prolog_stream(+*Module, +Mode, -Stream, +Options*)

Create a new stream that implements its I/O by calling predicates in *Module*. The called predicates are:

Module : `stream_write`*(+Stream, +String)*

Called for a `Mode = write` stream if data is available. *String* contains the (textual) data that is written to *Stream*. The callback is called if the buffer of *Stream* overflows, the user calls `flush_output(Stream)` or *Stream* is closed and there is buffered data.

Module : `stream_read`*(+Stream, -Term)*

Called for a `Mode == read` stream to get new data. On success the stream extracts text from the provided *Term*. *Term* is typically a string, atom, code or character list. If term is not one of the above, it is handed to `writeq/1`. To signal end-of-file, unify stream with an empty text, e.g., `stream_read(Stream, "")`.

Module : `stream_close`*(+Stream)*

Called when the stream is closed. This predicate must succeed. The callback can be used to cleanup associated resources.

The current implementation only deals with text streams. The stream uses the `wchar_t` encoding. The buffer size must be a multiple of `wchar_t`, i.e., a multiple of four for portability. The *newline* mode of the stream is `posix` on all platforms, disabling the translation `"\n" --> "\r\n"`.

Arguments

Options is currently ignored.

bug Futher versions might require additional callbacks. As we demand all callbacks to be defined, existing code needs to implement the new callbacks.

NetBSD Crypt license

```
* Copyright (c) 1989, 1993
*      The Regents of the University of California.  All rights reserved.
*
* This code is derived from software contributed to Berkeley by
* Tom Truscott.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. Neither the name of the University nor the names of its contributors
*    may be used to endorse or promote products derived from this software
*    without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
```

Index