SWI-Prolog C-library

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: `J.Wielemaker@cs.vu.nl`

March 28, 2011

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 Cywin. 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 Unix Process manipulation library

The `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 are modelled closely after their native Unix counterparts. Higher-level primitives, especially to make this library portable to non-Unix systems are desirable. Using these primitives and considering that process manipulation is not a very time-critical operation we anticipate these libraries to be developed in Prolog.

fork(*-Pid*)

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/2` is a simple direct interface to it.

exec(*+Command(...Args...)*)

Replace the running program by starting *Command* using the given commandline arguments. Each command-line argument must be atomic and is converted to a string before passed to the Unix call `execvp()`.

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***)**

Wait for a child to change status. Then report the child that changed status as well as the reason. *Status* is unified with `exited`(*ExitCode*) if the child with pid *Pid* was terminated by calling `exit()` (Prolog `halt/[0,1]`). *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***)**

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.[1].

pipe(-*InSream*, -*OutStream***)**

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***)**

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.

[1] kill/2 should support interrupt-names as well

detach_IO

This predicate is intended to create Unix deamon-processes. It preforms two actions. First of all, the I/O streams `user_input`, `user_output` and `user_error` are closed and rebound to a Prolog stream that returns end-of-file on any attempt to read and starts writing to a file named `/tmp/pl-out.pid` (where ⟨*pid*⟩ is the process-id of the calling Prolog) on any attempt to write. This file is opened only if there is data available. This is intended for debugging purposes.[2] Finally, the process is detached from the current process-group and its controlling terminal.

3 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 implements 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 multibyte 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()
- 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

[2]More subtle handling of I/O, especially for debugging is required: communicate with the syslog deamon and optionally start a debugging dialog on a newly created (X-)terminal should be considered.

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`.

  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(); 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)

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'], []).
```

Errors process_error(*Exe*, Status) where Status is one of exit(Code) or killed(Signal). Raised if the process does not exit with status 0.
To be done The detach options is a no-op.

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.

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`.

4 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.

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

bug Setting times does not work on Windows.

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 unerlying 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]*

True when *RelPath* is a relative path to AbsPath, relative to *RelTo*. *Path* and *RelTo* are first handed to `absolute_file_name/2`, which makes the absolute **and** canonical. Below is an example:

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

Parameters

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.

5 Socket library

The `socket` library 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:

1. 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.

2. Use multiple threads, handling either a single blocking socket or a pool using I/O multiplexing as above.

3. Using XPCE's class *socket* which synchronises socket events in the GUI event-loop.

tcp_socket(*-SocketId*)
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*)
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:

- After `tcp_accept/3`, the server does a `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, -InStream, -OutStream*)
Open two SWI-Prolog I/O-streams, one to deal with input from the socket and one with output to the socket. If `tcp_bind/2` has been called on the socket. *OutSream* is useless and will not be created. After closing both *InStream* and *OutSream*, the socket itself is discarded.

tcp_bind(*+Socket, ?Port*)
Bind the socket to *Port* on the current machine. This operation, together with `tcp_listen/2` and `tcp_accept/3` implement the *server*-side of the socket interface. 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(*+Socket, +Backlog*)
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 suggested default value is 5.

tcp_accept(*+Socket, -Slave, -Peer*)
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(*+Socket, +Host:+Port*) *[deprecated]*
Connect *Socket*. After successful completion, `tcp_open_socket/3` can be used to create I/O-Streams to the remote socket. New code should use `tcp_connect/4`, which can be hooked to allow for proxy negotiation.

tcp_connect(+*Socket, +Host:+Port, -Read, -Write***)**

Client-interface to connect a socket to a given *Port* on a given *Host*. *Port* is either an integer or the name of a registered service. The fragment below connects to the `http://www.swi-prolog.org` using the service name instead of the hardcoded number '80'.

```
    Adress = 'www.swi-prolog.org':http,
    tcp_socket(Socket),
    tcp_connect(Socket, Adress, Read, Write),
```

This predicate can be hooked by defining the multifile-predicate socket:tcp_connect_hook/4. This hook is specifically intented for proxy negotiation. The code below shows the structure of such a hook. The predicates `proxy/1` and `proxy_connect/3` must be provided by the user.

```
:- 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).
```

tcp_setopt(+*Socket, +Option***)**

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.

nodelay

Same as `nodelay`(*true*)

nodelay(*Bool***)**

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`.

dispatch(*Bool***)**

In GUI environments (using XPCE or the Windows plwin.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***)**

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

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

As of SWI-Prolog 3.2.4, handling of non-blocking stream is supported. An attempt to read from a non-blocking stream 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_host_to_address(?*HostName*, ?*Address***)**

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 a `ip/4` term, it is resolved by `gethostbyaddr()` and the canonical hostname is unified with *HostName*.[3]

gethostname(-*Hostname***)**

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

5.1 Server applications

The typical sequence for generating a server application is defined below:

```
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)
```

[3]This function should support more functionality provided by `gethostbyaddr()`, probably by adding an option-list.

```
                    ]),
        dispatch(AcceptFd).

process_client(Socket, Peer) :-
        setup_call_cleanup(tcp_open_socket(Socket, In, Out),
                           handle_service(In, Out),
                           close_connection(In, Out)).

close_connection(In, Out) :-
        close(In, [force(true)]),
        close(Out, [force(true)]).

handle_service(In, Out) :-
        ...
```

5.2 Client applications

The skeleton for client-communication is given below.

```
create_client(Host, Port) :-
        setup_call_catcher_cleanup(tcp_socket(Socket),
                                   tcp_connect(Socket, Host:Port),
                                   exception(_),
                                   tcp_close_socket(Socket)),
        setup_call_cleanup(tcp_open_socket(Socket, In, Out),
                           chat_to_server(In, Out),
                           close_connection(In, Out)).

close_connection(In, Out) :-
        close(In, [force(true)]),
        close(Out, [force(true)]).

chat_to_server(In, Out) :-
        ...
```

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

5.3 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.[4]

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

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

```
        close(Out),
        delete_stream_from_pool(In).
```

5.4 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).

    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.

6 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
```

	Parameters
Components	is a term uri_components(Scheme, Authority, Path, Search, Fragment). 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

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.

```
uri_is_global(URI) :-
        uri_components(URI, Components),
        uri_data(Components, scheme, Scheme),
        nonvar(Scheme).
```

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).
```

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` or `path`. Besides alphanumerical characters, the following characters are passed verbatim (the set is split in logical groups according to RFC3986).

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

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

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.

7 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).
```

7.1 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.

8 MIME decoding library

MIME (Multipurpose Internet Mail Extensions) is a format for serializing multiple typed data objects. It was designed for E-mail, but it is also used for other applications such packaging multiple values using the HTTP POST request on web-servers. Double Precision, Inc. has produced the C-libraries rfc822 (mail) and rfc2045 (MIME) for decoding and manipulating MIME messages. The `mime` library is a Prolog wrapper around the rfc2045 library for decoding MIME messages.

The general name 'mime' is used for this library as it is anticipated to add MIME-creation functionality to this library.

Currently the mime library defines one predicate:

mime_parse(*Data, Parsed*)

Parse *Data* and unify the result to *Parsed*. *Data* is one of:

stream(*Stream*)
: Parse the data from *Stream* upto the end-of-file.

stream(*Stream, Length*)
: Parse a maximum of *Length* characters from *Stream* or upto the end-of-file, whichever comes first.

Text
: Atoms, strings, code- and character lists are treated as valid sources of data.

Parsed is a tree structure of `mime`(*Attributes, Data, PartList*) terms. Currently either *Data* is the empty atom or *PartList* is an empty list.[5] *Data* is an atom holding the message data. The library automatically decodes *base64* and *quoted-printable* messages. See also the `transfer_encoding` attribute below.

PartList is a list of `mime/3` terms. *Attributes* is a list holding a subset of the following arguments. For details please consult the RFC2045 document.

type(*Atom*)
: Denotes the Content-Type, how the *Data* should be interpreted.

transfer_encoding(*Atom*)
: How the *Data* was encoded. This is not very interesting as the library decodes the content of the message.

character_set(*Atom*)
: The character set used for text data. Note that SWI-Prolog's capabilities for character-set handling are limited.

language(*Atom*)
: Language in which the text-data is written.

id(*Atom*)
: Identifier of the message-part.

description(*Atom*)
: Descrptive text for the *Data*.

disposition(*Atom*)
: Where the data comes from. The current library only deals with 'inline' data.

name(*Atom*)
: Name of the part.

filename(*Atom*)
: Name of the file the data should be stored in.

9 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 10

[5] It is unclear to me whether a MIME note can contain a mixture of content and parts, but I believe the answer is 'no'.

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[6] 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:

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

$1$qdaDeDZn$ZUxSQEESEHIDCHPNc3fxZ1
```

Encrypted is always an ASCII string. *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.

10 SHA1 and SHA2 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).

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.

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

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 integers representing the hash. The conversion is controlled by Options:

algorithm(+*Algorithm***)**

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

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.

10.1 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.

11 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 riscs or naming conflicts associated with temporary-file management. They do assume proper memory management by the hosting OS and cannot be used to pass data to external processes using a file-name.

There is no limit to the number of memory streams, nor the size of them. However, memory-streams cannot have multiple streams at the same time (i.e. cannot be opened for reading and writing at the same time).

These predicates are first of all intended for building higher-level primitives. See also `sformat/3`, `atom_to_term/3`, `term_to_atom/2` and the XPCE primitive `pce_open/3`.

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 currently one of `read` or `write`. The resulting *Stream* must be closed using `close/1`.

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 it 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 it 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.

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*.

12 Time and alarm library

The `time` provides timing and alarm functions.

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 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`.

13 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:

`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 catched 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).
```

NetBSD Crypt license

Index