Reference Manual for the
Elk UNIX Extension

Oliver Laumann

1. Introduction

This reference manual describes the primitive procedures and record types defined by the UNIX extension to Elk.

The UNIX extension provides easy access to most widely available UNIX system calls and C library functions from within Scheme programs. The extension supports a wide range of different UNIX platforms without restricting its functionality to the lowest common denominator or the POSIX 1003.1 functions. To simplify writing portable Scheme programs, the extension attempts to hide differences between the types of supported UNIX flavors. For example, programmers do not have to deal with the idiosyncrasies of the *wait()*, *waitpid()*, *wait3()*, and *wait4()* system calls or the *mktemp()*, *tmpnam()*, and *tempnam()* functions.

The UNIX extension defines procedures for low-level, file-descriptor-based I/O; creation of pipes; file/record locking; file and directory system calls; process creation and control; signal handling; error handling; and obtaining information about date, time, users, limits, process resources, etc. Terminal control is not yet supported by the current version.

The reference manual assumes that you are familiar with the most common UNIX system calls and C library functions; this document does not attempt to duplicate the contents of the standard UNIX documentation. Detailed descriptions are provided for functions that differ from the standard UNIX semantics.

2. Using the UNIX Extension

The UNIX extension is loaded by evaluating

```
(require 'unix)
```

in the interactive toplevel or in a Scheme program.

This causes the files *unix.scm* and *unix.o* to be loaded into the interpreter (*unix.o* has to be linked with the interpreter on platforms that do not support dynamic loading of object files). In addition, the *record* extension is automatically loaded if it is not yet present. The record extension is documented in a separate reference manual.

Loading the UNIX extension causes the features *unix* and *unix.o* to be provided. Optionally, one or more of the following features (described below) may be provided by the extension to indicate that certain UNIX features are available:

```
unix:wait-options
unix:wait-process
unix:record-locks
unix:file-locking
unix:symlinks
unix:reliable-signals
```

3. Record Types

Several procedures return their results as Scheme records. All record types defined by the UNIX extension are stored in variables with names of the form *<something>-record* (such as *system-record* or *passwd-record*). In addition, a type predicate, a record constructor, and accessor functions for all record fields are defined for each record type. For example, a *system* record type with the fields *hostname*, *sysname*, and *osname* is defined, resulting in variable *system-record* holding the record type descriptor, and the functions

```
(system-record? obj)
(make-system-record)

(system-hostname system-record)
(system-sysname system-record)
(system-osname system-record)
```

Use *define-record-modifiers* if you need the modifier functions for any of the records as well (see the record extension reference manual for details).

The following record types are defined by the UNIX extension:

Record Type	Fields
`stat`	`type mode ino dev nlink uid gid size atime mtime ctime`
`time`	`seconds minutes hours day-of-month month year weekday day-of-year dst`
`nanotime`	`nanoseconds minuteswest dst`
`system`	`hostname sysname osname`
`passwd`	`name password uid gid gecos homedir shell`
`group`	`name password gid members`
`resources`	`user-time system-time (...)`
`lock`	`exclusive? whence start length`
`wait`	`pid status code core-dump? resources`

4. Error Handling

The default error behavior of the primitive procedures defined by the UNIX extension is to invoke the standard Scheme error handler if a UNIX system call or library function fails. As an alternative, if a specific error action is to be performed by the application, a primitive procedure can be invoked under control of the *unix-errval* form. In this case, a unique *error object* is returned if a UNIX function signals an error. The standard UNIX system error message and the UNIX error number are made available to the application in any case. Details of the error handling facilities are described in the section "Error Functions" below.

5. Conventions

In the following sections, the names of procedure arguments can dictate that the arguments are of specific types. If an argument name is also the name of a Scheme data type, the argument must be an object of that type. For example, a procedure with an argument named *string* must be invoked with a string. File descriptor arguments (named *fdescr*, or $fdescr_1$, $fdescr_2$, etc.) and arguments named *length* are always non-negative integers; filename arguments (*filename*) are strings or symbols; and arguments with the suffix "?" are booleans.

6. Low-Level I/O, File Descriptors

(unix-close *fdescr***)** **procedure**

The UNIX *close()* system call. *unix-close* returns the non-printing object.

(unix-dup *fdescr***)** **procedure**
(unix-dup $fdescr_1$ $fdescr_2$**)** **procedure**

unix-dup invokes the *dup()* (first form) or *dup2()* (second form) system call. The result is a new file descriptor (an integer).

(unix-open *filename flags***)** **procedure**
(unix-open *filename flags mode***)** **procedure**

The UNIX *open()* system call. *flags* is a list of one or more symbols specifying the bitmask argument of the *open()* system call.

At least the flag symbols *read*, *write*, *append*, *create*, *truncate*, and *exclusive* are supported; additional symbols (such as *ndelay*) may be permitted on certain platforms. The procedure *unix-list-open-modes* can be used to obtain the list of flag symbols that are supported (see below). If *create* is present in the *flags* argument, the *mode* argument (an integer) must be supplied. At least one of the symbols *read* or *write* must be present in *flags*.

unix-open returns a new file descriptor (an integer).

Example:

```
(let ((f1 (unix-open "/etc/passwd" '(read))
      (f2 (unix-open "temp" '(read write create truncate) #o666))))
  ...)
```

(unix-list-open-modes) **procedure**

This procedure returns the list of *flag* symbols for the *unix-open* procedure that are supported on the local platform.

(unix-lseek *fdescr offset whence***)** **procedure**

The UNIX *lseek()* system call. *offset* is an integer; *whence* is one of the symbols *set*, *current*, and *end*. *unix-lseek* returns the new file position as an integer.

(unix-pipe) **procedure**

The *pipe()* system call. *unix-pipe* returns two file descriptors as a pair of integers.

(unix-read-string-fill! *fdescr string***)** **procedure**
(unix-read-string-fill! *fdescr string length***)** **procedure**

The *read()* system call. *unix-read-string-fill* invokes *read()* with the Scheme string as input buffer and the length of the string argument (first form) or the length supplied as a third argument (second form). If *length* is specified, it must be an integer between 0 and the length of *string*.

unix-read-string-fill! destructively overwrites the contents of the *string* argument. It returns the number of characters actually read (0 on EOF).

(unix-write *fdescr string***)** **procedure**
(unix-write *fdescr string length***)** **procedure**

The *write()* system call. For a description of the arguments see *unix-read-string-fill!* above. *unix-write* returns the number of characters actually written.

(unix-close-on-exec *fdescr***)** **procedure**
(unix-close-on-exec *fdescr on?***)** **procedure**

unix-close-on-exec returns the value of the *close-on-exec* flag for the given file descriptor as a boolean. If invoked with a second argument, the procedure sets the *close-on-exec* flag to the specified value and returns the previous value.

(unix-filedescriptor-flags *fdescr***)** **procedure**
(unix-filedescriptor-flags *fdescr flags***)** **procedure**

unix-file-descriptor-flags obtains the flags currently active for the given file descriptor (by means of the *fcntl()* system call) and returns them as a list of symbols. If invoked with a second arguments (a list of symbols), the procedure sets the flags to that argument and returns the previous value.

At least the flag symbol *append* is supported; additional symbols (such as *ndelay* or *sync*) may be permitted on certain platforms. The procedure *unix-list-filedescriptor-flags* can be used to obtain the list of file descriptor flags that are supported (see below).

Example:

```
;;; Enable non-blocking I/O for file descriptor (assumes POSIX)

(define (set-non-blocking fd)
  (let ((flags (unix-filedescriptor-flags fd)))
    (unix-filedescriptor-flags fd (cons 'nonblock flags))))
```

(unix-list-filedescriptor-flags) **procedure**

This procedure returns the list of file descriptor *flag* symbols that can be returned and set by *unix-filedescriptor-flags* on the local platform.

(unix-num-filedescriptors) **procedure**

unix-num-filedescriptors returns the maximum number of file descriptors per process in the local system. Depending on the UNIX flavor, the procedure invokes *getdtablesize()* or *sysconf()* or uses a static (compile-time) limit.

(unix-isatty? *fdescr***)** **procedure**

Returns #t if the specified file descriptor points to a terminal device, #f otherwise (the UNIX *isatty()* library function).

(unix-ttyname *fdescr***)** **procedure**

The UNIX *ttyname()* function. Returns the name of a terminal device as a string, or #f if the file descriptor is not associated with a terminal.

(unix-port-filedescriptor *port***)** **procedure**

This procedure returns the file descriptor associated with the file pointer conveyed in the specified Scheme port. An error is signaled if the port has already been closed or if it is a string port. *unix-port-filedescriptor* invokes the UNIX *fileno()* library function.

Manipulating a file descriptor obtained by *unix-port-filedescriptor* can cause unexpected interactions with the standard Scheme I/O functions and with the stdio buffering mechanism. In particular, it is not a good idea to close the file descriptor associated with the Scheme system's current input port or current output port.

Example:

```
(let ((stdout-fileno
        (unix-port-filedescriptor (current-output-port))))
  (if (unix-isatty? stdout-fileno)
      (begin
        (display (unix-ttyname stdout-fileno))
        (newline))))
```

(unix-filedescriptor–>port *fdescr type***)** **procedure**

Creates a Scheme port with a file pointer containing the specified file descriptor. *unix-filedescriptor–>port* is based on the *fdopen()* stdio function. *type* is a string and is used as the second argument for *fdopen()*.

The type of the newly created Scheme port is determined by the *type* argument. If *type* begins with the character #\r, an input port is created; #\w and #\a indicate an output port. If the second character of *type* is #\+ (*update*), an input-output (bidirectional) port is created.

No filename is associated with a Scheme port created by a call to *unix-filedescriptor–>port*. Instead, the string *unix-filedescriptor[%d]* (where *%d* is replaced by the numerical value of the file descriptor) will be returned by calls to *port-file-name* and displayed when printing the port.

Note that the file descriptor is closed by the garbage collector when the Scheme port becomes inaccessible.

7. Files and Directories

(unix-stat *file***)** **procedure**

The UNIX *stat()*/*fstat()* system call. *file* is either a filename or a file descriptor.

unix-stat returns a *stat-record* with the following fields:

Field	Type	Contents
`type`	symbol	file type
`mode`	integer	file access mode
`ino`	integer	inode number
`dev`	integer	device number
`nlink`	integer	number of links to file
`uid`	integer	file owner's user-ID
`gid`	integer	file owner's group-ID
`size`	integer	file size
`atime`	integer	last access time
`mtime`	integer	last modified time
`ctime`	integer	last inode change time

The file type is one of the symbols *directory*, *character-special*, *block-special*, *regular*, *symlink*, *socket*, *fifo*, or *unknown*.

(unix-access? *filename mode***)** **procedure**

unix-access? is based on the *access()* system call. *mode* is a list of zero or more of the symbols *read*, *write*, and *execute*. The empty list can be used to test for existence of the file. The procedure returns #t if the specified access is granted, #f otherwise.

(unix-chdir *filename***)** **procedure**

The UNIX *chdir()* system call. *unix-chdir* returns the non-printing object.

(unix-chmod *filename mode***)** **procedure**

The UNIX *chmod()* system call. *mode* is an integer. *unix-chmod* returns the non-printing object.

(unix-chown *filename uid gid***)** **procedure**

The UNIX *chown()* system call. *uid* and *gid* are integers. *unix-chown* returns the non-printing object.

(unix-unlink *filename***)** **procedure**

The UNIX *unlink()* system call. *unix-unlink* returns the non-printing object.

(unix-link $filename_1$ $filename_2$**)** **procedure**

The UNIX *link()* system call. *unix-link* returns the non-printing object.

(unix-rename $filename_1$ $filename_2$**)** **procedure**

The UNIX *rename()* system call. *unix-rename* returns the non-printing object.

On platforms where the *rename()* function is not available, the operation is performed by the equivalent sequence of *link()* and *unlink()* calls with interrupts disabled (certain restrictions apply in this case, e. g. directories cannot be renamed).

(unix-mkdir *filename mode***)** **procedure**

The UNIX *mkdir()* system call. *mode* is an integer. *unix-mkdir* returns the non-printing object.

(unix-rmdir *filename***)** **procedure**

The UNIX *rmdir()* system call. *unix-rmdir* returns the non-printing object.

(unix-utime *filename***)** **procedure**
(unix-utime *filename atime mtime***)** **procedure**

The UNIX *utime()* function. *unix-utime* sets the last access and last modification time of the given file to the current time (first form) or to the specified times (second form). *atime* and *mtime* are integers. *unix-utime* returns the non-printing object.

(unix-read-directory *filename***)** **procedure**

This procedure returns the contents of the specified directory as a list of filenames (strings). *filename* must be the name of a directory. *unix-read-directory* is based on the *opendir()*, *readdir()*, and *closedir()* functions.

Example:

```
;;; Return directory contents as list of (filename . filetype) pairs

(define (get-files-and-types directory)
  (map
    (lambda (file)
      (cons file (stat-type (unix-stat file))))
    (unix-read-directory directory)))

(pp (get-files-and-types "."))
```

(unix-tempname) **procedure**
(unix-tempname *directory***)** **procedure**
(unix-tempname *directory prefix***)** **procedure**

unix-tempname returns a pathname that can be used as the name of a temporary file (typically in /tmp or /usr/tmp). The newly created pathname is not the name of an existing file.

directory (a string or symbol) can be used to specify the directory component of the pathname; *prefix* (string or symbol), if present, may be used as a prefix for the filename component of the pathname. However, both arguments may be ignored by *unix-tempname*.

unix-tempname is based on one of the UNIX functions *tempnam()*, *mktemp()*, and *tmpnam()* (in that order); if none of these functions is available, an algorithm similar to the one employed by UNIX *mktemp()* is used.

8. Symbolic Links

The following procedures are only defined on platforms that support symbolic links. In this case, the feature *unix:symlinks* is provided when the UNIX extension is loaded.

(unix-lstat *filename***)** **procedure**

The UNIX *lstat()* system call. *unix-lstat* returns a *stat-record* (see *unix-stat* above).

(unix-readlink *filename***)** **procedure**

The UNIX *readlink()* system call. *unix-readlink* returns the contents of specified symbolic link as a string.

(unix-symlink *filename*$_1$ *filename*$_2$**)** **procedure**

The UNIX *symlink()* system call. *unix-symlink* returns the non-printing object.

9. File and Record Locking

The procedures described in this section are only defined if some form of file-based locking is available on the local system (either locking of entire files, or record locking). In this case, the feature *unix:file-locking* is provided at the time the UNIX extension is loaded. If the local system supports locking of individual file segments, the feature *unix:record-locks* is provided as well, and the locking primitives are based on the *fcntl()* system call (otherwise the *flock()* system call is used).

(unix-set-lock *fdescr lock wait?***)** **procedure**

The *lock* argument is a *lock-record* with these fields:

Field	Type	Contents
`exclusive?`	**boolean**	**exclusive lock (write lock) if #t, shared lock (read lock) otherwise**
`whence`	**symbol**	*set*, *current*, **or** *end*: **interpretation of** *start* **(see** *unix-lseek***)**
`start`	**integer**	**relative offset in bytes**
`length`	**integer**	**length in bytes (0 means lock to EOF)**

If record locks are supported, the fields *whence*, *start*, and *length* specify a segment in the file referred to by *fdescr* that is to be locked or unlocked. If only entire files can be locked, the contents of these fields are ignored by the lock procedures.

An arbitrary number of shared locks for a file or file segment may be active at a given time, but more than one exclusive lock, or both shared and exclusive locks, cannot be set at the same time. *fdescr* must be opened for reading to be able to set a shared lock; it must be opened with write access for an exclusive lock. A shared lock may be upgraded to an exclusive lock, and vice versa. Mandatory locking may or may not be supported by the local system.

If the *wait?* argument is #t and the specified lock cannot be applied, *unix-set-lock* blocks until the lock becomes available.

unix-set-lock returns #t if the specified lock could be applied, #f otherwise.

(unix-remove-lock *fdescr lock***)** **procedure**

This procedure removes the specified file lock or record lock from the file pointed to by *fdescr*. *lock* is a *lock-record*; see *unix-set-lock* above for a description. *unix-remove-lock* returns the non-printing object.

(unix-query-lock *fdescr lock***)** **procedure**

If record locks are not supported, this procedure always returns #f. If record locks are supported, *unix-query-lock* returns information about the first lock that would cause a call to *unix-set-lock* with *lock* to fail or block, or #f if no such lock exists (i.e. if claiming the specified lock would succeed). Information about the lock is returned as a pair; the car is an integer (the process-ID of the the process that owns the lock), the cdr is a *lock-record*. The process-ID may be meaningless in a network environment.

10. Obtaining Password and Group File Entries

The procedures defined in this section are used to obtain entries from the system's *passwd* and *group* databases.

(unix-get-passwd) **procedure**
(unix-get-passwd *user***)** **procedure**

If invoked without an argument, this procedure returns the next entry from the *passwd* database. Successive calls to *unix-get-passwd* return entries in a random order. The *user* argument, if present, is either the login name of a user (a string or symbol) or a numeric user-ID (an integer). In this case, the *passwd* entry for this user is returned.

unix-get-passwd returns a *passwd-record* with the following fields:

Field	Type	Contents
`name`	**string**	**login name**
`password`	**string**	**login password**
`uid`	**integer**	**numeric user-ID**
`gid`	**integer**	**numeric primary group-ID**
`gecos`	**string**	**contents of GECOS field**
`homedir`	**string**	**home directory of user**
`shell`	**string**	**login shell of user**

unix-get-passwd is based on the UNIX *getpwent()*, *getpwuid()*, and *getpwnam()* functions.

(unix-get-group) **procedure**
(unix-get-group *group***)** **procedure**

unix-get-group is identical to *unix-get-passwd* (see above), except that the system's *group* database is used instead of the *passwd* database.

The result value is a *group-record* with these fields:

Field	Type	Contents
`name`	**string**	**group's name**
`password`	**string**	**group's password**
`gid`	**integer**	**numeric group-ID**
`members`	**list of symbols**	**group members**

unix-get-group is based on the UNIX *getgrent()*, *getgrgid()*, and *getgrnam()* functions.

(unix-rewind-passwd) **procedure**
(unix-rewind-group) **procedure**

These procedures rewind the *passwd* and *group* files by calling the *setpwent()* and *setgrent()* UNIX functions.

(unix-end-passwd) **procedure**
(unix-end-group) **procedure**

unix-end-passwd and *unix-end-group* close the *passwd* and *group* files by calling the UNIX functions *endpwent()* and *endgrent()*.

11. Process Creation and Control

(unix-system *string***)** **procedure**

unix-system starts ''/bin/sh'' as a child process with *string* as input and waits until the shell terminates. All file descriptors except standard input, standard output, and standard error output are closed in the child process. *unix-system* returns the exit code of the shell as an integer or, if the shell was interrupted by a signal, the termination status as a list of one integer element. If the shell could not be executed, exit code 127 is returned.

(unix-open-input-pipe *string***)** **procedure**
(unix-open-output-pipe *string***)** **procedure**

The UNIX *popen()* function. Both procedures create a pipe between the caller and a shell executing the command *string*; they return a Scheme port containing the file pointer associated with the pipe. Closing the Scheme port, or running the garbage collector after the port has become unused, causes the pipe to be closed by a call to the *pclose()* function.

unix-open-input-pipe returns an input port that can be used to read from the standard output of the specified command; *unix-open-output-pipe* returns an output port that accepts input to be sent to the standard input of the command.

(unix-fork) **procedure**

The UNIX *fork()* system call. *unix-fork* returns the process-ID of the newly created process as an integer in the parent process, and the integer 0 in the child process.

The child process, as its first action, invokes the *onfork handlers* that may have been registered by other Elk extensions that are currently active (one purpose of *onfork* handlers is to make new links to temporary files in the newly created child process).

(unix-exec *filename arguments***)** **procedure**
(unix-exec *filename arguments environment***)** **procedure**

(unix-exec-path *filename arguments***)** **procedure**
(unix-exec-path *filename arguments environment***)** **procedure**

These procedures are based on the UNIX *execv()* family of system calls and library functions. The first argument is the name of the file to be executed. *arguments* is a list of strings to be passed to the program as arguments. The *environment* argument, if present, is a list of environment variable definitions to be used as the new program's environment. Each element of the list is pair of strings; the car of an element is the name of an environment variable, the cdr is the variable's value (the *unix-environ* primitive can be used to obtain the current environment of the running program).

unix-exec-path searches the specified filename in a list of directories obtained from the calling program's PATH environment variable. The variant of *unix-exec-path* that accepts an *environment* argument is not available on the currently supported platforms (the reason is that there is no *execvpe()* variant of the *execvp()* function, although *execve()*/*execle()* variants of *execv()* and *execl()* usually exist in UNIX).

unix-exec and *unix-exec-path* remove the temporary files used by the dynamic loading module of the interpreter kernel and invoke the finalization functions that may have been registered by extensions. As a result, attempting to load an object file after a call to *unix-exec* or *unix-exec-path* has returned (i. e. failed) may not work correctly. The finalization functions are only invoked once.

(unix-wait) **procedure**
(unix-wait *options***)** **procedure**

(unix-wait-process *pid***)** **procedure**
(unix-wait-process *pid options***)** **procedure**

unix-wait and *unix-wait-process* are based on the UNIX *wait()* family of system calls and library functions. Both procedures return a *wait-record* with the following fields:

Field	Type	Contents
`pid`	**integer**	**process-ID of the terminated child process**
`status`	**symbol**	**reason for process termination**
`code`	**integer**	**exit code or termination status (signal)**
`core-dump?`	**boolean**	**#t if a core-dump was produced**
`resources`	**resources-record**	**resources of terminated process**

See *unix-process-resources* below for a description of the *resources-record* type.

The *wait-record* result holds the process-ID and termination status of one of the terminated (or stopped) children of the calling process. The value of the *status* is one of the symbols *stopped* (if the child process has been stopped), *signaled* (child process is terminated due to a signal), or *exited* (child process has invoked *exit()*). *code* holds the exit code (if *status* is *exited*), or a signal number (if *status* is either *stopped* or *signaled*). The *resources* field holds the user and system time consumed by the child process and its children in nanoseconds (additional resources may be supplied in future versions). The fields of the *resources* record are #f on platforms that do not support the *wait3()* or *wait4()* system call.

unix-wait-process allows to collect the termination status of an individual process or a group of processes specified by the integer *pid* argument. This procedure is only defined on platforms where the *waitpid()* or *wait4()* system call is available. In this case, the feature *unix:wait-process* is provided when the UNIX extension is loaded.

If no child process is available (or, in case of *unix-wait-process*, no process as specified by the *pid* argument), the *pid* field in the result is set to -1, and the *status* field is set to the symbol *none*.

The *options* argument, if present, is a list of one or more of the symbols *nohang* and *untraced*. Options are only supported if the feature *unix:wait-options* is provided.

(unix-process-resources) **procedure**

This procedure is based on the UNIX *times()* library function. *unix-process-resources* returns the resource usage of the calling process and its terminated children as a pair of *resources-records*. Each *resources-record* has the following fields:

Field	Type	Contents
`user-time`	**integer**	**user time in nanoseconds**
`system-time`	**integer**	**system time in nanoseconds**

Addition fields may be supplied in future versions.

(unix-environ) **procedure**

unix-environ returns the program's environment as a list of pairs. The car of each element is the name of an environment variable (a string), the cdr is the value of that variable (a string).

(unix-getenv *string***)** **procedure**

This procedure returns the value of the environment variable with the name *string* as a string, or #f if the specified variable is not defined.

(unix-working-directory) **procedure**

unix-working-directory returns the calling program's current working directory as a string. The procedure is based on the *getcwd()* or *getwd()* function if any of these is available and invokes the ''pwd'' command otherwise.

(unix-getlogin) **procedure**

unix-getlogin returns the login name as a string (obtained by the UNIX *getlogin()* library function).

(unix-getuids) **procedure**
(unix-getgids) **procedure**

unix-getuids (*unix-getgids*) returns the calling program's real and effective user-IDs (group-IDs) as a pair of integers.

(unix-getpids) **procedure**

unix-getpids returns the process-ID of the calling process and the parent process-ID as a pair of integers.

(unix-getgroups) **procedure**

unix-getgroups returns the current supplementary group-IDs of the process as a list of integers.

Example:

```
;;; Get list of names of supplementary group-IDs

(define (get-group-names)
  (map
    (lambda (gid)
      (group-name (unix-get-group gid)))
    (unix-getgroups)))
```

(unix-umask *mask***)** **procedure**

The UNIX *umask()* system call. *mask* is an integer. The procedure returns the previous value of the umask.

(unix-nice *incr***)** **procedure**

The UNIX *nice()* function. *incr* is an integer. *unix-nice* returns the new nice value (or zero on some platforms).

(unix-sleep *seconds***)** **procedure**

The UNIX *sleep()* function. *seconds* is a positive integer. The procedure returns the non-printing object.

12. Obtaining System Information

(unix-system-info) **procedure**

This procedure returns a *system-record* with these fields:

Field	Type	Contents
`hostname`	**string**	**the system's hostname**
`sysname`	**string**	**type of hardware platform**
`osname`	**string**	**operating system type and version**

The hostname is determined by a call to the UNIX *gethostname()* or *uname()* function; the system name and OS name are obtained from the configuration file that has been used to configure and install Elk.

(unix-file-limit *limit file***)** **procedure**

unix-file-limit can be used to query various system limits and options associated with files. *limit* is a symbol identifying the type of limit; *file* is a filename or file descriptor.

At least the following limits and options can be queried:

Limit/Option	Meaning
`max-links`	**maximum number of links to a file or directory**
`max-name`	**maximum length of a filename**
`max-path`	**maximum length of a pathname**
`pipe-buf`	**pipe buffer size**
`no-trunc`	**filename exceeding maximum length causes error instead of being silently truncated**

Additional limits may be present on some platforms. The list of limits actually supported by this procedure can be obtained by a call to *unix-list-file-limits* (see below).

If present, the POSIX *pathconf()*/*fpathconf()* function is used to query a limit; in this case the specified filename or file descriptor is supplied as an argument to *pathconf()* or *fpathconf()*. If *pathconf()* is not available, or if calling it is not appropriate for the type of limit, a static (compile-time) value is returned.

The result type of *unix-file-limit* depends on the type of the specified limit (boolean in case of *no-trunc*, integer otherwise).

(unix-list-file-limits) **procedure**

This procedure returns the list of limit symbols that can be supplied as arguments to *unix-file-limit* (see above).

(unix-job-control?) **procedure**

This predicate returns #t if UNIX job control is available on the local system, #f otherwise. In a POSIX environment, this procedure may call *sysconf()*.

13. Date and Time

(unix-time) **procedure**

The UNIX *time()* function. *unix-time* returns the number of seconds elapsed since midnight UTC, January 1, 1970 (*The Epoch*) as an integer.

(unix-nanotime) **procedure**

This procedure returns the number of nanoseconds elapsed since The Epoch as an integer. *unix-nanotime* invokes one of the UNIX functions *gettimeofday()*, *ftime()*, *time()* (in that order, depending on which of these function is available), thus providing up to microsecond resolution.

(unix-decode-localtime *time***)** **procedure**
(unix-decode-utc *time***)** **procedure**

Both procedures convert the specified time (a number of seconds as returned by *unix-time*) into a *time-record*; *unix-decode-localtime* corrects for the local time zone and DST adjustment (based on the UNIX *localtime()* and *gmtime()* functions).

A *time-record* has the following fields:

Field	Type	Range
`seconds`	**integer**	**0..61**
`minutes`	**integer**	**0..59**
`hours`	**integer**	**0..23**
`day-of-month`	**integer**	**1..31**
`month`	**integer**	**0..11**
`year`	**integer**	**(year - 1900)**
`weekday`	**integer**	**0..6**
`day-of-year`	**integer**	**0..365**
`dst`	**integer**	**1 if DST in effect**

Example:

```
;;; Return date as a string of the form "Nov 3, 1993"

(define (date-string)
  (let* ((months "JanFebMarAprMayJunJulAugSepOctNovDec")
         (time (unix-decode-localtime (unix-time)))
         (month-inx (* 3 (time-month time))))

    (format #f "~a ~a, ~a"
            (substring months month-inx (+ 3 month-inx))
            (time-day-of-month time) (+ 1900 (time-year time)))))
```

(unix-time->string *time***)** **procedure**

This procedure converts the specified time into a string; it is based on the *ctime()* and *asctime()* UNIX functions. *time* is either an integer (number of seconds) or a *time-record*.

14. Signals

The procedures described in this section (except *unix-kill*, *unix-list-signals*, and *unix-pause*) are only defined if the local system supports reliable signals (either BSD-style or POSIX signals). In this case, the feature *unix:reliable-signals* is provided when the UNIX extension is loaded.

(unix-kill *pid signal***)** **procedure**

The UNIX *kill()* system call. *pid* is an integer; *sig* is either an integer (a signal number) or a symbol (a signal name). At least the following signal names are supported:

Signal names		
`sigalrm`	`sigbus`	`sigfpe`
`sighup`	`sigill`	`sigint`
`sigkill`	`sigpipe`	`sigquit`
`sigsegv`	`sigterm`	

The list of signal names actually supported by the local system can be obtained by calling *unix-list-signals* (see below).

unix-kill returns the non-printing object.

(unix-list-signals) **procedure**

This procedure returns a list of signal names (symbols) that are supported by the system.

(alarm *seconds***)** **procedure**

The UNIX *alarm()* function. *seconds* is a positive integer. *unix-alarm* returns the number of seconds remaining from the previously set alarm.

(unix-pause) **procedure**

The UNIX *pause()* function. This procedure does not return.

(unix-signal *sig action***)** **procedure**
(unix-signal *sig***)** **procedure**

unix-signal defines or queries the action to be performed when a signal is delivered to the program. If an *action* argument is specified, this action is associated with the signal *sig*, and the previous action for this signal is returned. If no action is given (second form), *unix-signal* just returns the action currently associated with *sig*.

sig is the name of a signal (see *unix-kill* for a description). The action associated with *sigbus*, *sigfpe*, *sigill*, *sigint*, *sigkill*, *sigsegv*, and *sigabrt* (if supported) cannot be altered; either because UNIX does not permit this (*sigkill*), or because the signal can be generated as the result of an internal fatal error (*sigbus* etc.), or because it is used by the interpreter internally (*sigsegv* is used by the incremental garbage collector). The action associated with the *interrupt* signal can be controlled by redefining the standard Elk *interrupt-handler* (see the Elk reference manual for details).

action can be one of the following:

the symbol *ignore*
: the specified signal is ignored

the symbol *default*
: the default action for this signal is established

the symbol *exit*

cleanup and exit: if the signal is delivered, the interpreter's temporary files are removed, the finalization functions and static C++ destructors of dynamically loaded extensions are invoked, and *exit()* is called with an exit code of 1

a compound procedure

the procedure (signal handler) is invoked on delivery of the specified signal.

The procedure specified as a signal handler must accept one or more arguments. When the signal is delivered, the procedure is invoked with the signal name (a symbol) as an argument. Signal handlers must not return (i. e. they must either exit or call a continuation). If a signal handler returns, a message is displayed and the *reset* primitive is called.

The signal specified as an argument to *unix-signal* is added to (removed from) the signal mask maintained by the interpreter, i. e. calls to the *disable-interrupts* primitive block the signal from delivery.

unix-signal returns the previous (current) action for the specified signal (a procedure or *ignore*, *default*, or *exit*) or the symbol *handler* to indicate that the signal is handled internally by the interpreter.

15. Miscellaneous Functions

(unix-getpass *string***)** **procedure**

unix-getpass displays *string* on standard output, reads a password, and returns the password as a string. The procedure invokes the UNIX *getpass()* function.

16. Error Functions

(unix-errval *expression***)** **syntax**

Normally, a Scheme error is signaled by the UNIX extension whenever a UNIX system call or library function invoked by any of the above primitives fails. The macro *unix-errval* allows an application to handle an error condition in a specific way without the need to redefine the standard error handler of Elk.

unix-errval evaluates the specified expression and returns the result of the evaluation. If, during evaluation of the expression, an error is signaled due to failure of a UNIX function, the corresponding primitive procedure returns a unique *error object* instead of performing normal error handling.

For example, evaluating the expression

```
(unix-close 1000)    ; close a bad file descriptor
```

would invoke the standard Scheme error handler in the normal way, whereas evaluating

```
(unix-errval (unix-close 1000))
```

would return an error object to allow the application to handle the error locally. Note that evaluation of the enclosing expression is not interrupted when an error is signaled, i. e. the expression

```
(unix-errval (begin (unix-close 1000) 5))
```

would return the integer 5.

(unix-error? *obj***)** **procedure**

This procedure returns #t if *obj* is the *error object*, #f otherwise. *unix-error?* is typically used to check whether a primitive invoked under control of *unix-errval* has signaled an error.

(unix-errno) **procedure**

Returns the UNIX *errno* set by the last system call that has failed. Error codes are represented as symbols corresponding to the names of the standard UNIX error numbers with letters converted to lower case, i. e. *enomem*, *ebadf*, etc. The exact set of error codes that can be returned is platform-dependent.

The value returned by *unix-errno* is not reset when a UNIX system call executes successfully. However, value of *unix-errno* is also affected by functions from the Elk kernel (such as *open-input-file*) and possibly other extensions that make use of system calls.

(unix-perror *string***)** **procedure**

unix-perror writes *string* followed by a colon and a short message describing the last UNIX error encountered to the current output port. *unix-perror* makes use of the "~E" format specifier of the format primitive.

17. Examples

This program implements a simple program interface to the UNIX *dc* desktop calculator command. The procedure *calc-open* starts the *dc* command and establishes two pipes to/from the child process; the procedure *calc* sends its argument (a *dc* expression as a string) as input to *dc*; *calc-close* closes the pipes and waits for the subprocess to terminate.

```
(require 'unix)

(define calc-from-dc)   ; input port: standard output of dc command
(define calc-to-dc)     ; output port: standard input of dc command
(define calc-dc-pid)    ; process-ID of child process running dc

(define calc-dc-command "/bin/dc")

(define (calc-open)
  (let* ((from (unix-pipe))
         (to (unix-pipe))
         (redirect-fd (lambda (a b)
                        (unix-dup a b) (unix-close a))))
    (set! calc-dc-pid (unix-fork))
    (if (zero? calc-dc-pid)
        (begin
          (unix-close (car from))
          (unix-close (cdr to))
          (redirect-fd (car to) 0)
          (redirect-fd (cdr from) 1)
          (unix-exec calc-dc-command '("dc")))
        (begin
          (unix-close (cdr from))
          (unix-close (car to))
          (set! calc-to-dc   (unix-filedescriptor->port (cdr to)   "w"))
          (set! calc-from-dc (unix-filedescriptor->port (car from) "r"))))))

(define (calc expr)
  (format calc-to-dc "~a~%" expr)
  (flush-output-port calc-to-dc)
  (read-string calc-from-dc))

(define (calc-close)
  (close-output-port calc-to-dc)
  (close-input-port calc-from-dc)
  (unix-wait-process calc-dc-pid))


;;; Test -- print sqrt(2):

(calc-open)
(display (calc "10k 2v p")) (newline)
(calc-close)
```

The following procedure copies a file; the arguments are the source and target file names. The second argument may name a directory, in this case the file is copied into the directory. The target file must not yet exist. *copy-file* preserves the access mode of the source file.

```
(require 'unix)

(define copy-buffer-size 8192)

(define (copy-file from to)
  (let ((from-stat (unix-stat from))
        (to-stat (unix-errval (unix-stat to))))

    (if (eq? (stat-type from-stat) 'directory)       ; complain if "from"
        (error 'copy-file "~s is a directory" from)) ;   is a directory

    (if (and (not (unix-error? to-stat))             ; destination exists
             (eq? (stat-type to-stat) 'directory))   ;   and is a directory?
        (set! to (format #f "~a/~a" to from)))

    (let* ((to-fd (unix-open to '(write create exclusive)
                             (stat-mode from-stat)))
           (from-fd (unix-open from '(read)))
           (buf (make-string copy-buffer-size)))

      (let loop ((num-chars (unix-read-string-fill! from-fd buf)))
           (if (positive? num-chars)
               (begin
                 (unix-write to-fd buf num-chars)
                 (loop (unix-read-string-fill! from-fd buf)))))

      (unix-close from-fd)
      (unix-close to-fd))))
```

lock-vi starts the *vi* editor with the specified file name. It provides exclusive access to the file during the editing session by applying a write lock to the file and removing it when the editor finishes. A message is displayed periodically if the lock is held by somebody else.

```
(require 'unix)

(define (lock-vi file)
  (let* ((fd (unix-open file '(read write)))
         (lock ((record-constructor lock-record) #t 'set 0 0)))

    (let loop ()
         (if (not (unix-set-lock fd lock #f))
             (begin
               (format #t "Someone else is editing ~s...~%" file)
               (unix-sleep 10)
               (loop))))

    (unix-system (format #f "vi ~a" file))
    (unix-remove-lock fd lock)))
```

pipe-size attempts to determine the capacity of a pipe. It creates a pipe, places the write end of the pipe into non-blocking I/O mode and writes into the pipe until it is full, counting the characters successfully written.

```
(require 'unix)

(define (pipe-size)
  (let* ((pipe (unix-pipe))
         (flags (unix-filedescriptor-flags (cdr pipe)))
         (len 32)                    ; assumes capacity is multiple of len
         (noise (make-string len)))

    ;; enable non-blocking I/O for write side of pipe:
    (unix-filedescriptor-flags (cdr pipe) (cons 'ndelay flags))

    (unwind-protect
      (let loop ((size 0))
           (if (unix-error? (unix-errval (unix-write (cdr pipe) noise)))
               (if (memq (unix-errno) '(eagain ewouldblock))
                   size
                   (error 'pipe-size "~E"))
               (loop (+ size 32))))
      (unix-close (car pipe))
      (unix-close (cdr pipe)))))
```

Index

V

W

Table of Contents