



e-POSIX

**The definitive and complete
Eiffel to Standard C and
POSIX 1003.1 binding**

written by Berend de Boer

Contents

1	Requirements and installation	1
1.1	Requirements	1
1.2	Compiling the C code	1
1.2.1	Compiling on Unix	1
1.2.2	Compiling on Windows	2
1.2.3	Library naming conventions	2
2	Using e-POSIX	4
2.1	Using <code>library.xace</code>	4
2.2	Gobo 3.3	5
2.3	Vendor specific notes	5
2.3.1	ISE Eiffel	5
2.3.2	SmartEiffel	5
2.3.3	Visual Eiffel	6
2.4	Platform specific notes	6
2.4.1	Linux	7
2.4.2	FreeBSD	7
2.4.3	Cygwin	7
2.4.4	BeOS	7
2.4.5	QNX	7
2.4.6	Solaris	8
2.4.7	Win32	8
3	Design notes	9
3.1	Why an entire reimplementaion?	9
3.2	Goals and guidelines	9
3.3	Class structure	10
3.4	Clients of this library	11
3.5	Forking	11
3.6	Books	14
4	Layers	15
4.1	Layers architecture	15
4.2	Standard C	15
4.3	Windows	15
4.3.1	Writing portable programs	15
4.3.2	Compiling POSIX programs in Windows	16
4.3.3	Native Windows	16
4.4	Introduction to the next chapters	18
5	Working with memory	19
5.1	Introduction	19
5.2	Allocating memory	19

5.3	Allocating memory	20
5.4	Using shared memory	20
5.5	Memory maps	21
6	Working with files	23
6.1	Introduction	23
6.2	Standard C notes	23
6.3	Compatibility with Gobo	23
6.4	Working with streams	23
6.5	Working with streams using Standard C only	28
6.6	Working with file descriptors	29
6.7	Windows systems: binary mode versus text mode	32
7	Working with files: advanced topics	34
7.1	Redirecting stderr to stdout	34
7.2	Talking to your modem	34
7.3	Non-blocking I/O	36
7.4	Asynchronous I/O	36
8	Working with the file system	38
8.1	Portability	38
8.2	Standard C	38
8.3	POSIX	40
9	Working with processes	45
9.1	Introduction	45
9.2	Executing a child command	45
9.3	Catching a signal with Standard C	46
9.4	Catching a signal with POSIX	48
9.5	General wait for child handler	49
9.6	Forking a child process	50
10	Querying the operating system	53
10.1	Current time	53
10.2	Accessing environment variables	54
10.3	Capabilities	55
11	Working with the network	56
11.1	Sockets	56
11.2	Echo client	56
11.3	Echo client and server	58
12	Working with the network: advanced topics	61
12.1	Introduction	61
12.2	FTP client	61
12.3	HTTP client	62
12.4	HTTP server	63
12.5	IMAP4 client	63
12.6	IRC client	64

12.7	SMTP client	65
13	Writing daemons	67
13.1	Introduction	67
13.2	Windows	67
13.3	Creating a daemon	67
13.4	Logging messages and errors	68
13.5	ULM based logging	69
14	Writing CGI programs	72
15	Error handling	80
15.1	Error handling with exceptions	80
15.2	Manual error handling	82
16	Security	85
16.1	Denial of service attacks	85
16.2	Authorization bypass attacks	86
17	Accessing C headers	87
17.1	Making C Headers available to Eiffel	87
17.2	Distinction between Standard C and POSIX headers	88
17.3	C translation details	88
A	Posix function to Eiffel class mapping list	90
	To do	96
	<i>ABSTRACT_DIRECTORY</i>	96
	<i>EPX_FILE_SYSTEM</i>	96
	<i>STDC_FILE</i>	96
	<i>STDC_LOCALE_NUMERIC</i>	96
	<i>STDC_PATH</i>	96
	<i>STDC_TIME</i>	96
	<i>POSIX_DAEMON</i>	96
	<i>POSIX_EXEC_PROCESS</i>	96
	<i>POSIX_FILE_DESCRIPTOR</i>	97
	<i>POSIX_MEMORY_MAP</i>	97
	<i>POSIX_SEMAPHORE</i>	97
	<i>POSIX_SIGNAL</i>	97
	<i>POSIX_STATUS</i>	97
	<i>POSIX_MQUEUE</i>	97
	Security	97
	Windows code	98
	Other	98
	Known bugs	98

Bibliography	99
Index	100

Introduction

It has been a great pleasure for me when I could announce the first public alpha release of this manual. And then came the betas and the first release. Writing libraries like this is boring stuff. Every Eiffel programmer should have had access to all those Standard C and POSIX routines long ago. Anyway, now you and me have. Whatever a C programmer can do, you can. And even more safe as this library protects you of inadvertently calling routines that are not portable (because they're simply not there :-)).

Writing libraries like this also seems to be a never ending story, as we now are at version 2.2. And my to do list hasn't shrunk, so stay tuned!

I actively support this library, so bug reports and wishes are gladly accepted. In the future, I hope to be able to expand this library to add more stuff from the Single Unix Specification such as `poll` and `select` support, and raw sockets. Also on my wish lists are an FTP, NEWS and IRC client implementation.

Have fun using this library and I like to hear about applications!

Licensing

This software is licensed under the Eiffel Forum Freeware License, version 2. This license can be found in the `forum.txt` file. Basically this license allows you to do anything with it, i.e. use it for commercial or Open Source software without restrictions. But don't sue me if something goes wrong. And give me some credits.

Also explicitly allowed is copying parts of this library to your own, for example copying certain Standard C or POSIX header wrappings. I prefer linking, but you don't have to retype everything if you don't want to link.

Support

e-POSIX is a fully supported program. You can send requests for help directly to me. But to help others profit from the discussion, and perhaps to get feedback when I'm short on time, it is suggested that support messages are sent to eposix@yahoogroups.com.

Latest versions and announcements are available from <http://groups.yahoo.com/group/eposix/>.

Commercial support

I'm available to give companies or organisations a one or two day course using POSIX and in particular this library. Prices are \$1000 NZD a day, excluding VAT, travel and hotel expenses. Contact me at berend@pobox.com.

Acknowledgements

I like to thank people who, one way or another, have helped me in creating this library. They're listed in order they have been involved with this library or manual:

- **Eugene Melekhov** <eugene_melekhov@object-tools.com>: compiled it with Visual Eiffel. As Visual Eiffel is the most strict compiler, he found a great many oversights that SmallEiffel didn't catch.
- **mico/E team**: I got many ideas for my C interface from the mico/E project. Sometime ago **Andreas Schulz** wrote me that the micoe team wanted to use e-POSIX in mico/E. Andreas also reported problems and suggested improvements, especially in the `EPX_CGI` class. Andreas and Robert Switzer, thanks for the bug reports!
- **Ida de Boer** <ida@gameren.nl>: it was she who provided you with the POSIX to Eiffel mapping table in [appendix A](#).
- **Steve Harris** <scharris@worldnet.att.net>: suggested improvements, found a CAT call problem and we had an interesting discussion about forking.
- **Jörgen Tegnér** <teg@post.netlink.se> reported a problem with an example, and a bug in `POSIX_EXEC_PROCESS`.
- **Marcio Marchini** <mqm@magma.ca> contributed a lot to e-POSIX. He gave very useful advice, submitted code, and supplied patches to compile e-POSIX better on Windows. I think it is fair to say that you thank the Windows support in e-POSIX to Marcio.
- **Eric Bezault**: I've had some insightful discussions with Eric regarding architecture of libraries such as e-POSIX. I think we never agreed :-), but the alternative error handling is due to his comments!
- **Andreas Leitner**: Discussions about using eposix which will lead to even closer integration with Gobo in subsequent releases.
- `[sven]`: various comments and suggestions.

Colophon

The text of this manual was entered with GNU Emacs 21.3.1 on Linux according to the LFS method. It was typeset with pdfTeX using the ConTeXt macro package, see <http://www.pragma-ade.com>. BON diagrams were created with METAPOST.

In this chapter:

1. Requirements
1.2 Compiling the C code

1

Requirements and installation

1.1 Requirements

e-POSIX has three requirements:

1. e-POSIX requires Gobo release 3.4 or higher. You can download Gobo at <http://www.gobosoft.com/>. Gobo must be installed.
2. e-POSIX requires that the environment variable `EPOSIX` is set to the root directory where the e-POSIX are unpacked.
3. On Windows, e-POSIX requires that the environment variable `GOBO_CC` is set to the name of the C compiler you are using. Failure to do so will result in link errors. Perhaps in a future geant release this will be set automatically.

1.2 Compiling the C code

Before e-POSIX can be used, a few C files need to be compiled into a library. The steps differ if you are using a Unix derivative, or a Windows based system.

1.2.1 Compiling on Unix

Before the C files can be compiled, e-POSIX must be configured. If you have just one Eiffel compiler on your system, this should be sufficient:

```
./configure --prefix=$EPOSIX
make
```

If you have multiple Eiffel compilers, you can specify the compiler with:

```
./configure --with-compiler=ve --prefix=$EPOSIX
```

The `-prefix` switch is a trick to make sure that you can type:

```
make install
```

after the make was successful. With this step the library is installed into the `\$EPOSIX/lib` directory. This is the location where e-POSIX's `src/library.xace` expects it. Without the `-prefix` switch the library will usually be installed in `/usr/local/lib`.

More information about `configure` options can be displayed with:

```
./configure --help
```


1.2.2 Compiling on Windows

For Windows system, I've supplied a tool —build with e-POSIX— that can build the necessary e-POSIX library for your Eiffel and C compiler.

Type:

```
makelib
```

to get help. Type:

```
makelib -ise -msc
```

to compile the C code with Microsoft's Visual C compiler targeting the ISE Eiffel compiler.

Only the Microsoft supplied library did work, i.e. link, with VisualEiffel:

```
makelib -ve -msc
```

Type:

```
makelib -se -bcb
```

to compile the C code with Borland's C compiler targeting SmartEiffel. It was tested with the free Borland C version 5.5 compiler.

Type:

```
makelib -se -lcc
```

to compile the C code with elj-win32's lcc C compiler.

If you have both the Borland C compiler and lcc installed, make sure the `make.exe` in your path is the correct one!

The generated library will have the name of the C compiler in its path. Make sure `GOBO_CC` has the correct value when compiling an e-POSIX program, see [table 1.1](#).

<code>bcb</code>	Borland C compiler.
<code>msc</code>	Microsoft C compiler.
<code>lcc</code>	lcc-win32 compiler.

Table 1.1 Possible values for the `GOBO_CC` environment variable

If you want to compile the `eposix` library for use in a multi-threaded application, pass the `-mt` switch to `makelib.exe`:

```
makelib -ise -bcb -mt
```

This is only supported for the ISE Eiffel compiler. `eposix` is not specifically written for use in multi-threaded programs nor tested much in such environments. There are certain areas (exit handling, signal handling) that are not multi-thread safe.

1.2.3 Library naming conventions

The name of this library starts with `libeposix`. On Unix the name of the Eiffel vendor is appended, so `libeposix_se.a` is the library for SmartEiffel. On Windows systems the name

of the Eiffel vendor and the C compiler are appended. On Windows different C compilers have incompatible libraries, so they need to be distinguished. On Windows the e-POSIX library for ISE Eiffel compiled with the Microsoft Visual C compiler is called `libposix_ise_msc.lib`.

The vendor names are derived from the names the Gobo Eiffel package uses, i.e. the `GOBO_EIFFEL` environment variable.

The C compiler is derived from the `GOBO_CC` environment variable.

In this chapter:

2. Using library.xace
2.2. Gobo 3.3
2.3. Vendor specific notes
2.4. Platform specific notes

2

Using e-POSIX

2.1 Using library.xace

Since Gobo 3.0 Eiffel library writers have a new great tool at their dispose: `gexace`. Eiffel library writers have to write and maintain just a single file, `library.xace`. You can this file file in the `e-POSIX src` subdirectory.

Typically, a `library.xace` is included in a `system.xace`. A typical example, including all required Gobo files, is:

```
<?xml version="1.0"?>

<system name="eposix_test">
  <description>
    system:      "eposix example program"
    author:      "Berend de Boer [berend@pobox.com]"
    copyright:   "Copyright (c) 2002-2003, Berend de Boer"
    license:     "Eiffel Forum Freeware License v2 (see forum.txt)"
    date:        "$Date: $"
    revision:    "$Revision: $"
  </description>

  <root class="{ROOT_CLASS}" creation="make"/>

  <option unless="{DEBUG}">
    <option name="assertion" value="none"/>
    <option name="garbage_collector" value="internal"/>
    <option name="finalize" value="true" unless="{GOBO_EIFFEL}=ve"/>
  </option>
  <option if="{DEBUG}">
    <option name="assertion" value="all"/>
    <option name="garbage_collector" value="internal"/>
    <option name="finalize" value="false"/>
  </option>

  <cluster name="example" location="{EPOSIX}/doc" unless="{GOBO_EIFFEL}=ve"/>
```

```

    <mount location="{EPOSIX}/src/library.xace"/>
    <mount location="{GOBO}/library/xml/library.xace"/>
    <mount location="{GOBO}/library/parse/library.xace"/>
    <mount location="{GOBO}/library/lexical/library.xace"/>
    <mount location="{GOBO}/library/structure/library.xace"/>
    <mount location="{GOBO}/library/kernel/library.xace"/>
    <mount location="{GOBO}/library/utility/library.xace"/>
    <mount location="{GOBO}/library/kernel.xace"/>

</system>

```

2.2 Gobo 3.3

e-POSIX can be used with the previous release of Gobo, Gobo 3.3. Do a:

```
make clean
```

in order to regenerate the provider parser and scanner files. You will need to copy two files from Gobo 3.4 (or CVS version):

1. `[KL_STRING_EQUALITY_TESTER]` class to `$GOBO/library/kernel/basic/`.
2. `[UC_STRING_EQUALITY_TESTER]` class to `$GOBO/library/kernel/unicode/`.

2.3 Vendor specific notes

2.3.1 ISE Eiffel

e-POSIX supports ISE Eiffel 5.5. e-POSIX has been tested under the following conditions:

1. I used Microsoft Windows 2000, Service Pack 2.
2. I used the Borland C 5.5 and Microsoft Visual C++ 6.0 compiler.

2.3.2 SmartEiffel

e-POSIX was tested with SmartEiffel 1.1 on FreeBSD, Linux, QNX, Solaris and Windows.

Because SmartEiffel has a tendency to provide lots of non-ELKS routines in its kernel classes—a bad thing in my opinion—I had to write a new `ANY`. My `ANY` renames `GENERAL.remove_file`, so I wouldn't get a conflict with `POSIX_FILE_SYSTEM.remove_file`.

There is no reason for the presence of `GENERAL.remove_file`, I expect this to be removed soon¹, so my `ANY` can be deleted when this has happened.

Some issues have been detected with SmartEiffel 1.1. If you use this version, I suggest you don't use boost mode and don't use the garbage collector. The most stable release of SmartEiffel seems to be -0.74.

¹ I wrote that two years ago. . .

If you use lcc-win32 as your C compiler, note that for the Gobo [XM_UNICODE_CHARACTER_CLASSES](#) class SmartEiffel generates code that does not compile with lcc-win32 due to some line length limit. This problem was still present with the latest lcc-win32 compiler, version 3.8, compiled on December 23.

If you use SmartEiffel and if you don't use Gobo's gexace tool to generate SmartEiffel's Ace file, you might see a complaint about a routine `stdc_signal_switch_switcher` not being found when linking. In that case you will need to put a `cecil.se` file in your directory. The contents of this file should be:

```
-- The name of our include C file:
cecil.h
-- The features called from C:
stdc_signal_switch_switcher STDC_SIGNAL_SWITCH switcher
stdc_exit_switch_at_exit STDC_EXIT_SWITCH at_exit
```

But I strongly suggest to make the switch to Gobo's gexace tool as this tool makes compilation for different Eiffel compilers a lot easier.

2.3.3 Visual Eiffel

e-POSIX has been tested with two of ObjectTool's offerings:

1. Their free VisualEiffel 4.1 for Linux.
2. VisualEiffel Professional 4.1 for Windows.
3. Version 4.0 or earlier are not supported, due to a change in semantics of the make routine of the STRING class. Only version 4.1 is ELKS compatible.

Follow these steps to compile with VisualEiffel 4 on Windows:

1. Make sure the `VE_BIN` environment variable is set to the Bin directory in the VisualEiffel sub-directory. On my system it is set to `M:/ProgramFiles/ObjectTools/VisualEiffel/Bin`.
2. Create the `libeposix_ve_msc.lib` library using the Microsoft Visual C compiler:

```
makelib -ve -msc
```
3. Use gexace to generate an `.esd` file.
4. Make sure to set the linker supplier option to Microsoft in your `system.xace` file! So an option like this should be present:

```
<option name="linker" value="microsoft" if="\${GOBO_EIFFEL}=ve"/>
```

2.4 Platform specific notes

Although e-POSIX should, in principle, run on every platform that supports Standard C or POSIX, it cannot be tested on every platform by me alone. This section gives details about the platforms I've used. The main thing you need to do is to edit e-POSIX's `src/library.xace` to the proper libraries for your platform are linked. The default `src/library.xace` is suited for Linux only.

2.4.1 Linux

The latest version of e-POSIX was tested with kernel 2.4.29 and glibc 2.2.93.

2.4.2 FreeBSD

The latest version of e-POSIX was tested with FreeBSD 4.11-STABLE. FreeBSD doesn't support `fdatasync`, so we do a `fsync` there. Cases like that are automatically detected by the `configure` script.

You have to edit `/src/library.xace` to link the proper library for FreeBSD. Look at the comments.

After a `make clean` you have to use `gmake` instead of `make`.

2.4.3 Cygwin

The latest version of e-POSIX was tested with Cygwin 1.3.x. Some remarks:

1. Locking doesn't seem to be supported.
2. `fifo`'s (`mkfifo`) are not supported.
3. No support for `fdatasync`, so we do a `fsync` there.

2.4.4 BeOS

The latest version of e-POSIX was tested with BeOS 5.03. BeOS has a nice POSIX compatibility layer. Some remarks:

1. Locking doesn't seem to be supported.
2. `fifo`'s (`mkfifo`) are not supported.
3. Hard links are not supported, only symbolic links.
4. No support for `fdatasync`, so we do a `fsync` there.
5. Sockets work in BeOS, but they are not file descriptors. Stick to the `EPX_SOCKET` classes like `EPX_TCP_CLIENT_SOCKET`. Never pass a socket to an `ABSTRACT_FILE_DESCRIPTOR` as that will not work.
The trick is that `read` and `write` in `EPX_SOCKET` call `recv` and `sendmsg`. If you pass a socket to an `ABSTRACT_FILE_DESCRIPTOR`, the POSIX `read` and `write` routines will be called.
6. BeOS does not support non-blocking i/o on file descriptors or sockets. e-POSIX says it does if you ask `ABSTRACT_FILE_DESCRIPTOR.supports_nonblocking_io`, but it doesn't.
BeOS has some options for non-blocking sockets, but they're very primitive and it seems you can't turn blocking off once it has been turned on for example.

2.4.5 QNX

The latest version of e-POSIX was tested with QNX 6.2.1.

You have to edit `/src/library.xace` to link the proper library for QNX. Look at the comments.

2.4.6 Solaris

e-POSIX was tested against Solaris 10 for Intel. Make sure to add the `-std=c99` option to `CFLAGS`. Solaris seems to require this if the `POSIX-1.2001` define is set.

You have to edit `/src/library.xace` to link the proper library for Solaris. Look at the comments.

2.4.7 Win32

The latest version of e-POSIX was tested with Windows 2000, Service Pack 2. On Win32, Standard C is fully supported. With e-POSIX's abstract layer, parts of POSIX and the Single Unix Specification are also supported. Support isn't as extensive as using the Cygwin tools.

In this chapter:

3.1 Why an entire reimplementation?
3.2 Goals and guidelines
3.3 Class structure
3.4 Clients of this library
3.5 Forking
3.6 Books

3

Design notes

3.1 Why an entire reimplementation?

One might wonder why I reimplemented the entire Standard C and POSIX library when most vendors also have classes that deal with files, the file system, signals and such. Unfortunately, these classes are not complete nor very portable between vendors. For someone who wants to compile against all the major vendors —and there are good reasons to do this— there is currently no portable solution. That's why many portable Eiffel programs more or less contain the same code again and again. There are some attempts to write more portable libraries, for example the [Unix File/Directory Handling Cluster](#) by Friedrich Dominicus, but they also are not complete nor is the implementation satisfactory. For example they usually have much logic at the C level. I wanted only C glue code: all intelligence should be in the Eiffel code.

Another attempt is done by the Gobo cluster: it attempts to provide users with a set of classes that work across all Eiffel vendors by using only the native facilities offered by each implementation. This approach has the advantage that no C compilation is necessary. The disadvantages are:

1. The contract for these classes is probably not specifiable: for which platforms and which assumptions are the contracts valid? Are these contracts the same in all implementations?
2. It is incomplete, i.e. it doesn't cover most of the POSIX routines.

That's why I started to make the entire Standard C and POSIX routines available to Eiffel programmers. All these routines are nicely wrapped in classes. I spend a lot of time designing and refactoring these, comments and improvements about its structure are very appreciated.

The advantage of making POSIX available to Eiffel programmers is that someone doesn't need to think about creating a set of portable file and directory classes that work on every known operating system. POSIX is available on many platforms and for other systems there either is an emulation or a POSIX mapping available. It's better to reuse that, instead of reinventing work that took years to complete.

3.2 Goals and guidelines

The goals and guidelines for this library were:

1. A complete Standard C implementation for those who didn't have access to POSIX routines.
2. A complete POSIX implementation.
3. Do the job in such a way that it will become the official Eiffel POSIX mapping.
4. All classes should satisfy the demands posed by the query-command separation principle.

5. The native Standard C and POSIX routines should be available to those who don't want to go through a certain class layer.
6. The names in use in the POSIX world like file descriptor or memory map are used as class names. This should make it easy to find a class if one knows the POSIX name.
7. If a command fails, an exception code is raised. This differs from the POSIX routines where one is expected to test for error and query the `errno` variable. The only exception is `unlink`: when the file does not exist, no exception is raised.
8. POSIX assumptions should be made explicit. For Eiffel this means specifying explicit pre- and postconditions.
9. Use of constants to influence the way a method should be avoided by providing clearly named methods. So instead of passing a constants to the `POSIX_FILE.open` function to open a file read-only, one can also call `open_read`.
10. Attempt to create non-deferred class that refer to an entity that exists in the POSIX world. Creation of an object is binding to that entity, or creation of that entity.
11. Names should be clear, and Eiffel-like. They should not differ in just one character. POSIX names are also made available to ease use of this library for programmers that know POSIX well.

3.3 Class structure

e-POSIX makes available all the Standard C and POSIX headers in classes like `CAPI_STDIO` and `PAPI_UNISTD`. More details about the header translation are in [chapter 17](#).

However, making the plain C API available is not a very interesting addition to an Eiffel programmer's toolkit. Therefore, this library's second attempt was to make an effective OO-wrapper, while making a careful distinction between what is available in the Standard C and what is available in POSIX. This distinction is reflected in e-POSIX's directory structure, see [figure 3.1](#).

The raw Standard C API is available in `src/capi`, the OO-wrapper is available in `src/standardc`. The raw POSIX API is available in `src/papi`, the OO-wrapper is available in `src/posix`.

Every Standard C and POSIX wrapper is derived from a common root, see also [figure 3.2](#):

1. If a class builds upon facilities available on Standard C, its name starts with the prefix `STDC_` and it inherits from `STDC_BASE`.
2. If a class builds upon facilities available in POSIX, its name starts with the prefix `POSIX_` and it inherits from `POSIX_BASE`.
3. If a class builds upon facilities available in the Single Unix Specification, its name starts with the prefix `SUS_` and it inherits from `SUS_BASE`. The support for the Single Unix Specification is not yet complete, but is continually enhanced.
4. Because we live in a world dominated by Microsoft Windows, and Microsoft Windows does not do POSIX, this would mean that many users only could use e-POSIX's Standard C facilities. These facilities are extremely limiting, for example there is no change directory command in Standard C. Therefore e-POSIX makes available an abstraction layer that covers routines that have an equivalent in POSIX and the Single Unix Specification. These classes start with the name `EPX_`. They always inherit from classes starting with `ABSTRACT_`. These abstract classes implement the common code. See [chapter 4.3.3](#) for more details.

Note that by using Cygwin you have a full POSIX emulation layer on Windows. In that specific environment you can use e-POSIX's entire POSIX and Single Unix Specification layer.

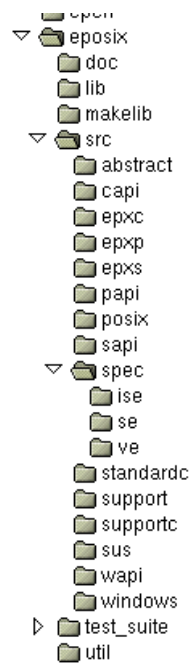


Figure 3.1 e-POSIX
directory structure

The wrapper classes should be fully command–query separated and use clear names. Often the POSIX name, if applicable, is also made available as an alias. If this is a good thing, I’m not sure. I hope it facilitates working with the wrapper classes if you already know POSIX.

Besides these directories, e-POSIX provides a number of extensions to the pure Standard C or POSIX routines. These can be found in the subdirectories that start with `src/epx`. A single letter indicates if the classes only built upon routines available in Standard C or POSIX:

1. `epxc`: Standard C based extensions like URI resolving, a MIME parser and XML generation.
2. `epxs`: Single Unix Specification based extension like an HTTP client.

3.4 Clients of this library

For client classes, two important classes are `STDC_CONSTANTS` and `POSIX_CONSTANTS`, see [figure 3.3](#). The wrapper classes tend to avoid having routines whose behavior drastically depends on passed constants. But if you need to use constants, your client class can just inherit from these classes and every Standard C and POSIX constant is available.

3.5 Forking

Implementing forking posed some interesting challenges. I started with the basic idea that every process has a pid:

```
class PROCESS
```

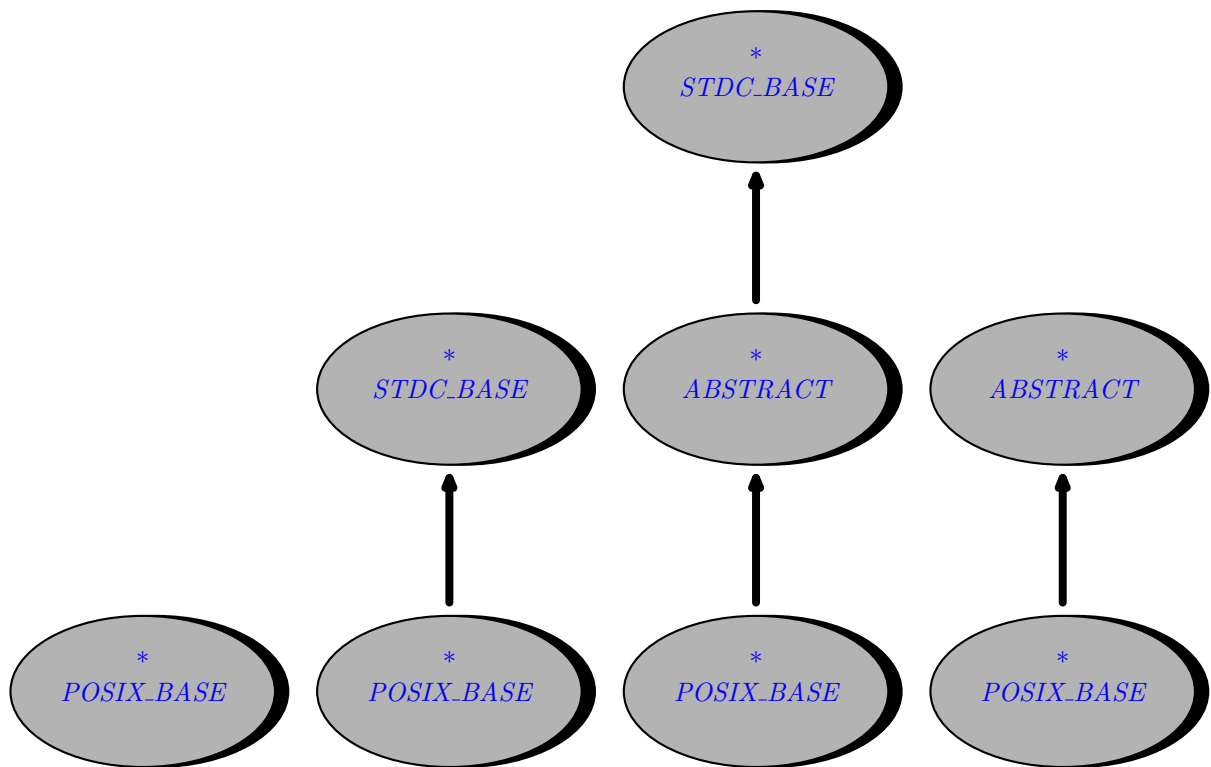


Figure 3.2 Inheritance structure

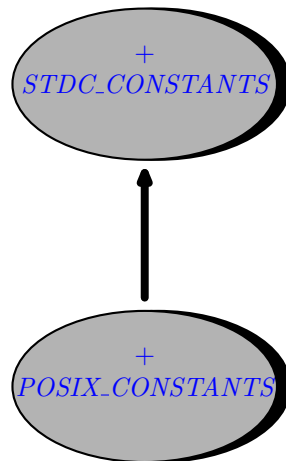


Figure 3.3 Standard C and POSIX constants

feature

pid: INTEGER

end

I wanted to be able to write two kinds of forking. The first one is forking a child as in:

```
class PARENT

inherit

    POSIX_CURRENT_PROCESS

feature

    make is
    local
        child: POSIX_CHILD_PROCESS
    do
        print ("My pid: ")
        print (pid)
        print ("%N")
        fork (child)
        print ("child's pid: ")
        print (child.pid)
        print ("%N")
        child.wait_for (True)
    end

end
```

However, I also wanted to fork myself, because that basically is what forking is!

```
class PARENT

inherit

    POSIX_CURRENT_PROCESS

    POSIX_CHILD_PROCESS

feature

    make is
    do
        fork (Current)
        wait
    end

    execute is
    do
        -- forked code
    end
```

end

The above code gives a name clash, because `POSIX_CURRENT_PROCESS.pid` is a call to the POSIX routine `getpid`, while the child's `pid` is a variable, which gets a variable after forking. You can solve this name clash yourself, but it is most easy to inherit from `POSIX_FORK_ROOT`, a clash which has solved this clash already.

If you fork a child, you must wait for it. For a child process, you can use `POSIX_CHILD.wait_for`, if you fork yourself, you must use `POSIX_CURRENT_PROCESS.wait`. The variable `waited_child_pid` will be set with the `pid` of the child process that `wait` waited for.

3.6 Books

Books that have been helpful during the development of e-POSIX where (POSIX1003.1, 1996, Plauger1991, 1991, Lewine1994, 1994), see the biography section at [page 99](#).

In this chapter:

4.1 Layers architecture

4.2 Standard C

4.3 Windows

4.4 Introduction to the next chapters

4

Layers

4.1 Layers architecture

e-POSIX is written in such a way that it is possible to write a pure Standard C based application (ANSI/ISO IS 9899: 1990), a pure POSIX application (Standard ISO/IEC-9945-1: 1990), or a pure Single Unix Specification version 3 application (http://www.unix-systems.org/single_unix_specification/). Although POSIX and the Single Unix Specification merged there specifications, they are still kept separate in e-POSIX, because the merge happened relatively recently and the pure POSIX functions are more very widely supported.

Based on these standards e-POSIX offers a compatibility layer. This layer offers a common framework for people that want to write code that works on both Unix and Windows systems. The compatibility layer uses all features that an operating system offers. If you use the network compatibility layer for example, you need a system that supports the Single Unix Specification.

4.2 Standard C

All Standard C classes start with STDC_. They are:

1. [STDC_TEXT_FILE](#): access text files.
2. [STDC_BINARY_FILE](#): access binary files.
3. [STC_TEMPORARY_FILE](#): create a temporary file, a file that is removed when it is closed or when the program terminates.
4. [STDC_CONSTANTS](#): access Standard C constants like error codes and such.
5. [STDC_BUFFER](#): allocate dynamic memory.
6. [STDC_ENV_VAR](#): access environment variables.
7. [STDC_FILE_SYSTEM](#): delete and rename files.
8. [STDC_SHELL_COMMAND](#): pass an arbitrary command to the native shell.
9. [STDC_SYSTEM](#): access information about the system the program is running on.
10. [STDC_CURRENT_PROCESS](#): access to current process related information like its standard input, output and error streams.
11. [STDC_TIME](#): access current time. Also can format a given time in various formats.

4.3 Windows

4.3.1 Writing portable programs

e-POSIX offers three alternatives to writing programs that run on both Unix and Windows platforms:

1. Write programs that only rely on Standard C. If you use only Standard C classes your program is probably quite portable. Standard C doesn't offer that much however.
2. Write programs that are based on POSIX. You use a POSIX emulator to compile and run your program unchanged on Windows. The only thing you have to be aware of is the distinction between binary and text files.
3. Write programs that are based upon e-POSIX's EPX_XXXX layer. This layer is based on e-POSIX's ABSTRACT_XXXX classes, that covers code that is common between Windows and a POSIX platform.

Previous versions of e-POSIX used a factory class approach to access this common code. This is no longer needed. The ABSTRACT_XXXX are made effective through EPX_XXXX classes when compiling for Windows or for POSIX.

The following sections offer more details about the last two approaches.

4.3.2 *Compiling POSIX programs in Windows*

You can also use a very large subset of POSIX under Windows with a POSIX emulator. I've tested this using SmartEiffel and Cygwin's freely available emulator. Here the steps:

1. Download the Cygwin toolkit from <http://sources.redhat.com/cygwin>.
2. Set the compiler in `compiler.se` to `gcc`. Leave the system in `system.se` to Windows.
3. Configure e-POSIX as described in 1.2 and create `libposix_se.a`

A few things are not available under Cygnus' POSIX emulation:

1. `POSIX_FILE_SYSTEM.create_fifo` is not supported. Any attempt to use it will return `ENOSYS`. I'm not sure if returning an error is the correct solution for applications that require POSIX compatibility, because you are only warned at run-time. Another solution would be to include a call to `mkfifo` and if you use it, let the linker complain.
2. There is no locking, so calls to `POSIX_FILE_DESCRIPTOR.get_lock` and such will fail.
3. Certain POSIX tests assume that a more Unix like environment is available, so not all tests will run. For example the standard Cygwin distribution doesn't have a `more` utility. If you make a symbolic link from `less` to `more` the child process test will run.
4. The current list of implemented functions is available from http://sources.redhat.com/cygwin/faq/faq_3.html#SEC17.

4.3.3 *Native Windows*

Previous versions of e-POSIX used a factory class approach to access Windows or POSIX specific code. This is obsolete.

If you want to write code that is portable between Windows and POSIX use the EPX_XXXX class layer. For example you can use the `EPX_FILE_DESCRIPTOR` to use file descriptors that are completely portable between these two OSes. Use `EPX_FILE_SYSTEM` to have access to file system specific code to change directories or get the temporary directory.

In general you can replace the `POSIX_` prefix with `EPX_` to compile most of the examples presented in the previous POSIX specific chapters. The classes currently available in the EPX_XXXX layer are:

- `EPX_CURRENT_PROCESS`.
- `EPX_EXEC_PROCESS`.
- `EPX_FILE_DESCRIPTOR`.
- `EPX_FILE_SYSTEM`.
- `EPX_PIPE`.

Figure one shows hoe the `EPX_FILE_DESCRIPTOR` class is derived from `ABSTRACT_FILE_DESCRIPTOR`. Both Windows and POSIX have an effective `EPX_FILE_DESCRIPTOR` class. Classes as `POSIX_FILE_DESCRIPTOR` implement POSIX specific functionality for a file descriptor.

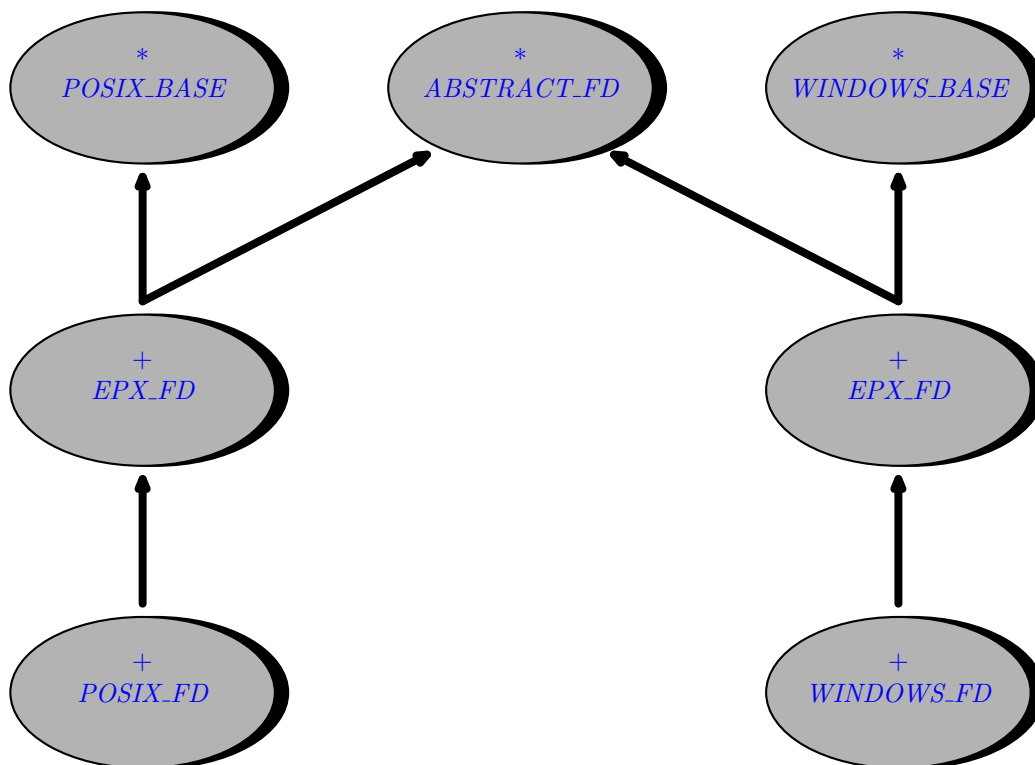


Figure 4.1 How EPX_XXXX classes are related to the POSIX and Windows classes

An example of using the `EPX_FILE_SYSTEM` class is shown below:

```

class EX_EPX1

inherit

    EPX_FILE_SYSTEM

creation


    make

feature
  
```



```
make is
local
  dir: STRING
do
  print ("Current directory: ")
  dir := current_directory
  print (dir)
  print ("%N")
  change_directory ("..")
  change_directory (dir)
  make_directory ("abc")
  rename_to ("abc", "def")
  remove_directory ("def")
end

end
```

In  all abstract classes are listed. There deferred features are made effective in the EPX class for the operating system you're compiling for.

4.4 Introduction to the next chapters

The following chapters are topic based: they discuss how to work with files for example and show examples for all layers and give hints what is and what isn't supported in each layer.

Instead of describing every class and every feature, I decided to show short and simple examples of common ways to use the various e-POSIX classes. Most examples assume a POSIX or Single Unix Specification environment. If you don't have POSIX available, you can try to replace the `POSIX_` prefix by `STDC_`. Most of the time the POSIX classes are based on the Standard C classes.

If you are looking for more examples, you might take a look at the classes in the `test_suite` directory. These classes should demonstrate and test almost every feature available in the POSIX classes.

In this chapter:

5.1 Introduction
5.2 Allocating memory
5.3 Allocating memory
5.4 Using shared memory
5.5 Memory maps

5

Working with memory

5.1 Introduction

e-POSIX has several classes that allocate memory. The main class is `STDC_BUFFER` (or the equivalent `POSIX_BUFFER`). This class allocates a memory block that isn't moved by the garbage collector. This is very useful for an Eiffel compiler that has a moving garbage collector.

You can also get access to shared memory using `POSIX_SHARED_MEMORY`.

5.2 Allocating memory

You can dynamically allocate memory with `STDC_BUFFER` which works just like `POSIX_BUFFER`.

class *EX_MEM2*

creation

make

feature

make is

local

mem: STDC_BUFFER

byte: INTEGER

do

create *mem.allocate_and_clear (128)*

mem.poke_uint8 (2, 57)

byte := mem.peek_uint8 (2)

mem.resize (256)

mem.deallocate

end

end

With the feature `STDC_BUFFER.allocate_and_clear` memory is allocated and cleared to all zeros.

`STDC_BUFFER` contains many routines to read bytes and strings from the memory it manages like `peek_int16`, `peek_uint16`, or `peek_int32`. It supports reading and writing 16 and 32 bit integers in little and big endian order with routines as `peek_int16_big_endian`, `peek_int16_little_endian`, and `poke_int32_big_endian`.

5.3 Allocating memory

Allocating dynamic memory is very useful, but not portably available for Eiffel programmers. With `POSIX_BUFFER` memory can be allocated, read and written to.

```
class EX_MEM
```

```
creation
```

```
    make
```

```
feature
```

```
    make is
```

```
        local
```

```
            mem: POSIX_BUFFER
```

```
            byte: INTEGER
```

```
        do
```

```
            create mem.allocate (256)
```

```
            mem.poke_uint8 (2, 57)
```

```
            byte := mem.peak_uint8 (2)
```

```
            mem.resize (512)
```

```
            mem.deallocate
```

```
        end
```

```
    end
```

For more information about the dynamic memory class, see [section 5.2](#).

5.4 Using shared memory

You can use shared memory to exchange data between different processes. It's dependent on your POSIX version if this is supported, so check for this capability explicitly!

```
class EX_SHARED_MEM1
```

```
inherit
```

```
    POSIX_SYSTEM
```

```
    POSIX_CURRENT_PROCESS
```

```
    POSIX_FILE_SYSTEM
```

creation*make***feature**

```

make is
  local
    fd: POSIX_SHARED_MEMORY
  do
    if not supports_shared_memory_objects then
      stderr.puts ("Shared memory objects not supported.%N")
      exit_with_failure
    end

    create fd.create_read_write ("/test.berend")
    fd.put_string ("Hello world.%N")
    fd.close
    unlink_shared_memory_object ("/test.berend")
  end
end

```

end

Make sure you always start a shared memory object with a slash. Else the behaviour is undefined or processes might not be able to find your shared memory.

There is not yet an abstract layer implementing shared memory, but you can use [WINDOWS_PAGING_FILE_SHARED_MEMORY](#) on Windows to get a similar effect.

5.5 Memory maps

You can map a file to memory using [POSIX_MEMORY_MAP](#).

```
class EX_MEMORY_MAP1
```

```
  inherit
```

```
    POSIX_SYSTEM
```

```
    POSIX_CURRENT_PROCESS
```

creation*make***feature**

```
make is
local
  fd: POSIX_FILE_DESCRIPTOR
  map: POSIX_MEMORY_MAP
  byte: INTEGER
  correct: BOOLEAN
do
  if supports_memory_mapped_files then

    -- Open a file.
    create fd.open_read_write ("ex_memory_map1.e")

    -- Create memory map.
    create map.make_shared (fd, 0, 64)

    -- Read a byte from the mapping.
    byte := map.peek_uint8 (2)
    correct := byte = ('a').code
    if not correct then
      print ("Oops.%N")
    end

    -- Cleanup.
    map.close
    fd.close
  end
end
end
```

There is no equivalent abstract layer class for memory mapping to support Windows yet.

In this chapter:

6.1 Introduction
6.2 Standard C notes
6.3 Compatibility with Gobo
6.4 Working with streams
6.5 Working with streams using Standard C only
6.6 Working with file descriptors
6.7 Windows systems: binary mode versus text mode

6 Working with files

6.1 Introduction

e-POSIX offers two different file classes: Standard C stream based and POSIX file descriptor classes. The main difference between stream and descriptor based classes is that the stream classes offer read and write caching. Output is not immediately written to disk or network for example.

6.2 Standard C notes

If you don't have access to a POSIX compatible system, you can use the underlying Standard C classes. Standard C is quite restricted in certain respects: you cannot change directories for example. On the other hand, this library gives you access to all Standard C routines, so you can use what's there and write an extremely portable program.

6.3 Compatibility with Gobo

Since version 2.0 e-POSIX is built upon foundations laid in Gobo. e-POSIX's `STDC_FILE/POSIX_FILE` and `ABSTRACT_FILE_DESCRIPTOR` are implementations of `KI_CHARACTER_INPUT_STREAM` and `KI_CHARACTER_OUTPUT_STREAM`.

The e-POSIX class `ABSTRACT_FILE_DESCRIPTOR` has support for non-blocking i/o, see [section 7.3](#). Gobo's `KI_CHARACTER_INPUT_STREAM` expects blocking i/o however. If you call `ABSTRACT_FILE_DESCRIPTOR.read_string` you will call the routine that has support for non-blocking i/o. Due to Eiffel's renaming mechanism, `ABSTRACT_FILE_DESCRIPTOR` will behave blocking when it is called as if it was a `KI_CHARACTER_INPUT_STREAM`.

6.4 Working with streams

The basic class for working with files, or streams as they are also called, is `POSIX_FILE`. There are two kinds of files: `POSIX_TEXT_FILE` and `POSIX_BINARY_FILE`. According to the POSIX standard, there is no distinction between binary and text files. But on certain systems you must use POSIX programs through an emulation layer. For example, on Windows Cygwin is a well-known POSIX emulator. To maintain compatibility with other Windows programs, Cygwin

distinguishes between text and binary files. If you use Cygwin to compile your POSIX programs, this distinction is therefore still important.

The first example shows how to open a text file, see also the corresponding BON diagram in [figure 6.1](#).

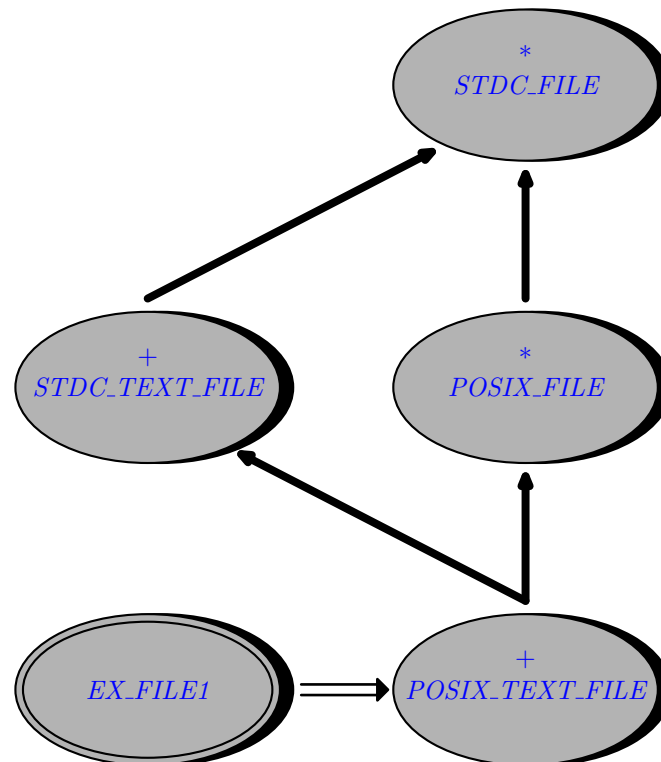


Figure 6.1 BON diagram of opening a text file.

```
class EX_FILE1
```

```
creation
```

```
make
```

```
feature
```

```
make is
```

```
local
```

```
file: POSIX_TEXT_FILE
```

```
do
```

```
create file.open_read ("/etc/group")
```

```
from
```

```
file.read_line
```

```
until
```

```
file.eof
```

```

loop
  print (file.last_string)
  print ("%N")
  file.read_line
end
file.close
end

```

end

It simply opens a file for reading and prints every line in it. Note that the line read does *not* include the end-of-line character. This is a change in behaviour from pre 2.0 eposix versions.

[POSIX_FILE] has two functions that read strings. These are `read_line` and `read_string`. `read_line` only returns when it has read an end-of-line character. If it has to read a 2GB characters to reach that, it will return a 2GB string. `read_string` returns a string with the given number of characters, or less if the end of the file is reached. These two functions have one other difference as well: `read_line` removes the end-of-line character(s), while `read_string` returns the raw string, including end-of-line characters and such.

At the end of the example, the file is closed. You don't need to explicitly close a file as it will be closed when your object is garbage collected. But I think it's a good thing not to rely or depend on this, but to close your external resources as soon as you're done using them. For example many systems have easily reached limits on the number of files a process can have open.

Reading binary files is almost the same loop, only you read it in chunks:

```
class EX_FILE2
```

creation

```
make
```

feature

```
chunk_size: INTEGER is 512
```

```
make is
```

```
local
```

```
file: POSIX_BINARY_FILE
```

```
buffer: POSIX_BUFFER
```

```
do
```

```
create file.open_read ("/bin/sh")
```

```
create buffer.allocate (chunk_size)
```

```
from
```

```
file.read_buffer (buffer, 0, chunk_size)
```

```
until
```

```
file.eof
```

```
loop
```

```
file.read_buffer (buffer, 0, chunk_size)
```



```

        end
        file.close
    end
end

```

end

This example uses a more safe version of buffer reading, `POSIX_FILE.read_buffer`. There is an untyped variant `POSIX_FILE.read` which accepts a pure pointer. There is no need to mention that you need to watch buffer overflows carefully with this last one!

Correctly looping through files, takes care. For example the following loop is wrong:

```
class EX_WRONG1
```

```
creation
```

```
    make
```

```
feature
```

```
    make is
```

```
        local
```

```
            file: POSIX_TEXT_FILE
```

```
        do
```

```
            create file.open_read ("/etc/group")
```

```
            from
```

```
            until
```

```
                file.eof
```

```
            loop
```

```
                file.read_string (256)
```

```
                print (file.last_string)
```

```
            end
```

```
            file.close
```

```
        end
```

end

After `POSIX_TEXT_FILE.read_string`, `eof` might be `True`. But the precondition for `last_string` is that `eof` is `false`. You will make an unnecessary extra loop. The correctly coded variant is:

```
class EX_WRONG2
```

```
creation
```

```
    make
```

```
feature
```

```
    make is
```

```
        local
```

```

    file: POSIX_TEXT_FILE
do
    create file.open_read ("etc/group")
from
until
    file.eof
loop
    file.read_string (256)
    if not file.eof then
        print (file.last_string)
    end
end
file.close
end

```

end

I myself prefer the first example, as the check is only in the **until** part, and not repeated in the loop. The following examples shows how a binary file is created and a string is written to it.

```

class EX_FILE3

inherit

    POSIX_FILE_SYSTEM

creation

    make

feature

    make is
    local
        file: POSIX_BINARY_FILE
    do
        create file.create_write (expand_path ("$/HOME/myfile.tmp"))
        file.put_string ("hello world.%N")
        file.close
    end

end

```

end

Depending on the platform you are running a backslash is turned into a slash or vice versa.

This example also demonstrates how path names —file and directory names— can be expanded: if you call `POSIX_FILE_SYSTEM.expand_path`, any environment variables in the path are expanded. Backslashes and slashes are always translated, but environment variable expansion has to be done explicitly.

You can move the file pointer with two different methods: `POSIX_FILE.seek` and `set_position`. The `seek` works with files up to 2 GB, `set_position` has no such limits. Use `tell` to get a position that can be passed to `seek`. Use `get_position` to get a position that can be passed to `set_position`.

class *EX_FILE5*

creation

make

feature

make is

local

file: POSIX_BINARY_FILE

pos1: INTEGER

pos2: STDC_FILE_POSITION

do

create *file.create_read_write* ("test.bin")

file.put_string ("one")

pos1 := file.tell

pos2 := file.get_position

file.put_string ("two")

file.seek (*pos1*)

-- or *file.set_position* (*pos2*)

file.read_string (3)

if not *file.last_string.is_equal* ("two") **then**

print ("unexpected read.%N")

end

file.close

end

end

6.5 Working with streams using Standard C only

Working with text files is equal to the POSIX classes, only you use the STDC prefix.

class *EX_FILE4*

creation

make

feature

```

make is
local
  file: STDC_TEXT_FILE
do
  create file.open_read ("/etc/group")
from
  file.read_line
until
  file.eof
loop
  print (file.last_string)
  print ("%N")
  file.read_line
end
file.close
end

end

```

Its BON diagram, see [figure 6.2](#) is therefore quite equal to the POSIX one, see [figure 6.1](#).

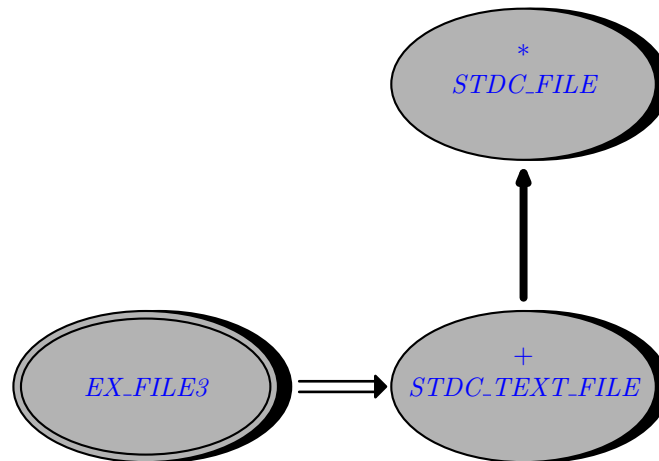


Figure 6.2 BON diagram of opening a Standard C text file.

6.6 Working with file descriptors

The file descriptors classes are quite equal to the file classes. The following example opens a file using `POSIX_FILE_DESCRIPTOR` and reads the first 64 bytes.

```
class EX_FDI
```

```
creation
```

```
make
```

feature

```

make is
  local
    fd: POSIX_FILE_DESCRIPTOR
  do
    create fd.open_read ("/etc/group")
    fd.read_string (64)
    print (fd.last_string)
    fd.close
  end
end

```

end

Unlike `POSIX_TEXT_FILE`, there is no easy way to detect end of line and end of file conditions. However, a file descriptor can easily be turned into a file as the following example demonstrates.

class EX_FD2**creation**

```
make
```

feature

```

make is
  local
    fd: POSIX_FILE_DESCRIPTOR
    file: POSIX_TEXT_FILE
  do
    create fd.open_read ("/etc/group")
    create file.make_from_file_descriptor (fd, "r")
    from
      file.read_string (256)
    until
      file.eof
    loop
      print (file.last_string)
      file.read_string (256)
    end
    file.close
    fd.close
  end
end

```

end

A file descriptor can also be used to lock, unlock or test for locks on a given file as the following example demonstrates. See also the accompanying BON diagram in [figure 6.3](#).

class EX_FD4

creation*make***feature***make is***local***some_lock,**lock: POSIX_LOCK**fd: POSIX_FILE_DESCRIPTOR***do****create** *fd.create_read_write ("test.tmp")**fd.put_string ("Test")***create** *lock.make**lock.set_allow_read**lock.set_start (2)**lock.set_length (1)**some_lock := fd.get_lock (lock)***if** *some_lock /= Void* **then***print ("There is already a lock?%N")***end***-- create exclusive lock**lock.set_allow_none**lock.set_start (0)**lock.set_length (4)**fd.set_lock (lock)**fd.close***end****end**

`POSIX_FILE_DESCRIPTOR.get_lock` is command–query separated, that is why it returns a new lock when queried and there is a lock. If there is no lock `get_lock` returns `Void`. The passed lock is not modified.

A file descriptor also gives you access to the attached terminal, if any. The following example demonstrates how to read a password without the password appearing on the screen.

class *EX_FD3***inherit***POSIX_CURRENT_PROCESS***creation**

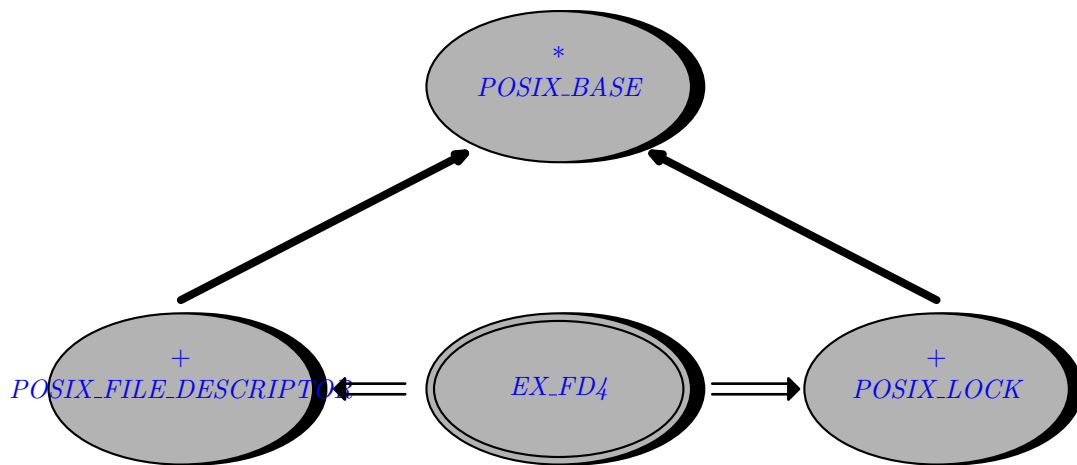


Figure 6.3 BON diagram of locking a portion of a file.

```

make
feature
  make is
  do
    print ("Password: ")
    stdout.flush

    -- turn off echo
    fd_stdin.terminal.set_echo_input (False)
    fd_stdin.terminal.apply_flush

    -- read password
    fd_stdin.read_string (256)

    -- turn echo back on
    fd_stdin.terminal.set_echo_input (True)
    fd_stdin.terminal.apply_now

    print ("%NYour password was: ")
    print (fd_stdin.last_string)
  end
end
  
```

6.7 Windows systems: binary mode versus text mode

If you are using Unix exclusively, you can skip this section.

Independent of what layer you use to write Windows programs, you have to deal with binary and text modes. And if you usually write Unix programs and want them to work on Windows too, you have to bother with it too.

On Windows, each line of a text files ends with a carriage return character followed by a line feed character. If you use a C text stream to read a file on Windows, a trick is employed: every occurrence of "%R%N" is replaced by a single "%N". If The same happens when writing to a text stream: you just have to write a single "%N" and the C run-time code replaces this by

So make sure you are using the proper classes if you use streams. Use `STDC_TEXT_FILE` if you want to read and write text files and use `STDC_BINARY_FILE` to read and write binary files.

File descriptors are binary only. So any descendant from `ABSTRACT_FILE_DESCRIPTOR` treats input and output as binary and does no translation whatsoever. If you use `ABSTRACT_FILE_DESCRIPTOR.read_line` to read lines, the end-of-line character may either be a "%R%N" or just a end-of-line characters regardless of the platform. So reading a file with Windows end-of-line characters on Windows or Unix will work exactly the same.

There is no explicit support for creating text files using file descriptors with the proper Windows end of file characters. Use either `STDC_TEXT_FILE` to create platform dependent end-of-lines or write the proper end-of-line characters yourself.

This discussion also applies to standard input and output. If you want to use binary standard input or binary standard output, use the file descriptors available in `EPX_CURRENT_PROCESS` as `fd_stdin` and `fd_stdout`. If you use `stdin` and `stdout` you can handle text files only on Windows. On Unix it does not matter.

For Cygwin users the story is somewhat more difficult it seems. File descriptors can be text or binary. The default is binary however. The following information can be helpful to get the binary versus text file distinction correct:

- Mount the volume in binary mode.
- Set the environment variable CYGWIN to 'binary'.

More information about Cygwin and CR/LF handling can be found at http://sources.redhat.com/cygwin/faq/faq_toc.html#TOC62.

In this chapter:

7.1 Redirecting stderr to stdout
7.2 Talking to your modem
7.3 Non-blocking I/O
7.4 Asynchronous I/O

7 *Working with files: advanced topics*

7.1 *Redirecting stderr to stdout*

If you want to redirect all output written by your program or any child you spawn to stdout, you can use the `POSIX_FILE_DESCRIPTOR.make_as_duplicate` call:

```
class EX_REDIRECTI
```

```
inherit
```

```
    POSIX_CURRENT_PROCESS
```

```
creation
```

```
    make
```

```
feature
```

```
    make is
```

```
    do
```

```
        -- flush stream buffers, else output may be in wrong order
```

```
        stdout.flush
```

```
        stderr.flush
```

```
        fd_stderr.make_as_duplicate (fd_stdout)
```

```
        -- all output written to stderr goes to stdout now
```

```
    end
```

```
end
```

It's a good idea to call this at the beginning of your program, before you have written anything to stderr or stdout. If you do that, you don't have to flush the stream buffers.

7.2 *Talking to your modem*

With e-POSIX you can talk to your modem. The implementation contains not all the details to write a full-featured program as minicom, but they will be added upon request.

The following example tries to talk to your modem —which is expected to be at `/dev/modem`— and queries its manufacturer.

```

class EX_MODEM

inherit

    POSIX_CURRENT_PROCESS

creation

    make

feature

    make is
    local
        modem: POSIX_FILE_DESCRIPTOR
        term: POSIX_TERMIOS
    do
        -- assume there is a /dev/modem device
        create modem.open_read_write ("/dev/modem")
        term := modem.terminal
        term.flush_input
        print ("Input speed: ")
        print (term.speed_to_baud_rate (term.input_speed))
        print ("%N")
        print ("Output speed: ")
        print (term.speed_to_baud_rate (term.output_speed))
        print ("%N")

        term.set_input_speed (B9600)
        term.set_output_speed (B9600)
        term.set_receive (True)
        term.set_echo_input (False)
        term.set_echo_new_line (False)
        term.set_input_control (True)
        term.apply_flush

        -- expect modem to echo commands
        modem.put_string ("AT%N")
        modem.read_string (64)
        print ("Command: ")
        print (modem.last_string)
        modem.read_string (64)
        print ("Response (expect ok): ")
        print (modem.last_string)
        modem.put_string ("ATIO%N")

```

```

    modem.read_string (64)
    print ("Command: ")
    print (modem.last_string)
    modem.read_string (64)
    print ("Response: ")
    print (modem.last_string)
    modem.close
end

end

```

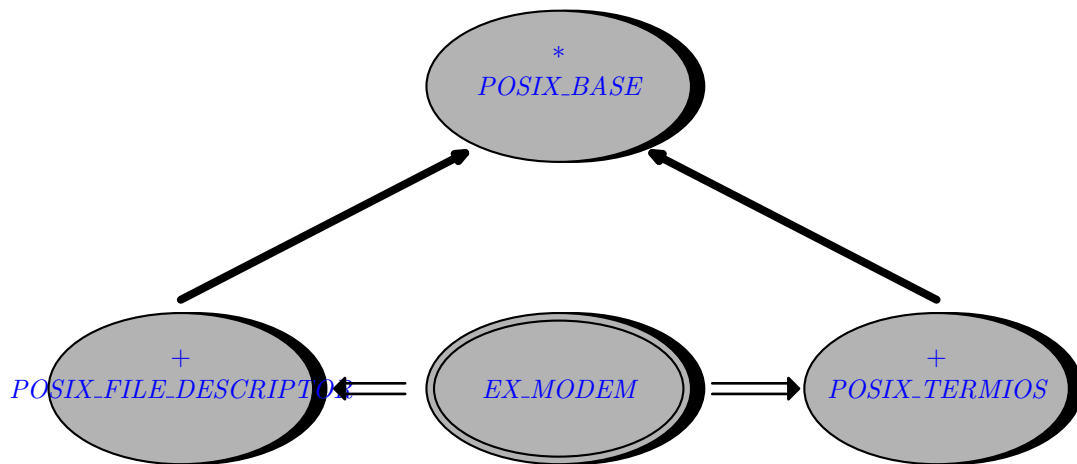


Figure 7.1 BON diagram of talking to a modem.

7.3 Non-blocking I/O

e-POSIX supports non-blocking i/o on its file descriptor classes, i.e. the descendants of `ABSTRACT_FILE_DESCRIPTOR`. Use `is_blocking_io` to query if the descriptor blocks on `read` or `write` if there is no data. Use `set_blocking_io` to change the behavior.

Use `supports_nonblocking_io` to query if the behavior with respect to blocking i/o can be changed. On Windows file i/o must be blocking. Only sockets on Windows can be non-blocking. On Unix all descriptors support non-blocking i/o.

See also [section 6.3](#) for non-blocking i/o when e-POSIX is used as a plugin for classes that expect a `KI_CHARACTER_INPUT_STREAM`. In such cases e-POSIX reverts to blocking i/o, even when non-blocking i/o has been enabled.

7.4 Asynchronous I/O

e-POSIX supports the asynchronous i/o features of POSIX. Not all Free Unices seem to support this feature, nor does their support seem to be error free.

Take a look at the following example:

```
class EX_ASYNC1
```

creation*make***feature**

```
make is
  local
    fd: POSIX_FILE_DESCRIPTOR
    request: POSIX_ASYNC_IO_REQUEST
  do
    create fd.create_read_write ("test.tmp")
    create request.make (fd)
    request.set_offset (0)
    request.put_string ("hello world.")
    request.wait_for
    fd.close
  end
```

end

The basic idea is that each asynchronous request is a separate object, modeled by `POSIX_ASYNC_IO_REQUEST`. You prepare it through calls like `set_buffer`, `set_count` and `set_offset`. You execute the request by calling `read` or `write`.

You can wait for the request to be complete by calling `wait_for`. It should be possible to force open requests to be synchronized to the disk with `synchronize`, but this does give strange results on Linux. So far I haven't got access to a machine that also implements asynchronous i/o to test if my code is correct.

In this chapter:

8.1 Portability
8.2 Standard C
8.3 POSIX

8

Working with the file system

8.1 Portability

Use the `EPX_` classes to write code that is portable between POSIX systems and Windows.

8.2 Standard C

Standard C doesn't offer much for file systems. You can only delete and rename files.

class *EX_DIR5*

inherit

STDC_FILE_SYSTEM

creation

make

feature

make is
do
rename_to ("qqtest.abc.tmp", "qqtest.xyz.tmp")
remove_file ("qqtest.xyz.tmp")
end

end

The BON diagram is shown in **figure 8.1**.

But you can manipulate filenames including directories, although technically they're not part of Standard C. The following example shows how filenames can be manipulated with `STDC_PATH`:

class *EX_FILENAME1*

creation

make

feature

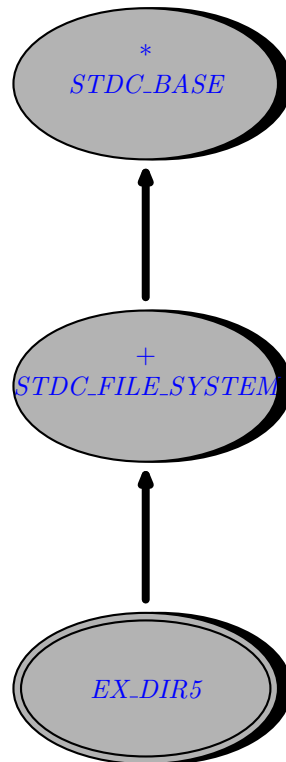


Figure 8.1 BON diagram of deleting and renaming files with Standard C.

```

make is
local
  path: STDC_PATH
do
  create path.make_from_string ("/tmp/myfile.e")
  path.parse (<<".e">>)
  print_path (path)

  create path.make_expand ("$/HOME/myfile.e")
  path.parse (<<".e">>)
  print_path (path)
end

print_path (a_path: STDC_PATH) is
do
  print ("Directory: ")
  print (a_path.directory)
  print ("", basename: ")
  print (a_path.basename)
  print ("", suffix: ")
  print (a_path.suffix)

```

```

    print ("%N")
end

```

end

The `parse` feature is used to parse a path into its components. Give it a suffix list to remove any matching suffices. Suffix matching is case-insensitive. If the suffix list is empty, no suffix matching will be done. This follows standard unix behaviour: if a filename has a dot in it, it does not necessarily mean that what follows after that dot is a suffix.

Create a path with `make_expand` to expand any environment variables in the given string to their values.

8.3 POSIX

POSIX defines many commands to navigate a file system. They're made available by the `POSIX_FILE_SYSTEM`. The following example navigates to the user's home directory, create a directory and removes it.

```

class EX_DIR1

inherit

    POSIX_FILE_SYSTEM

creation

    make

feature

    make is
    do
        change_directory (expand_path ("~"))
        make_directory ("qctest.xyz.tmp")
        remove_directory ("qctest.xyz.tmp")
    end

end

```

end

To get access to the file system, inheriting from the `POSIX_FILE_SYSTEM` class is easiest.

There are also lots of functions to test for existence, readability or writability of files. Use `is_modifiable` to test if a file is readable and writable.

```

class EX_DIR2

inherit

    POSIX_FILE_SYSTEM

```

creation*make***feature***make is***local***perm: POSIX_PERMISSIONS***do***print_info (is_existing ("/tmp"), "existing")**print_info (is_executable ("/bin/ls"), "executable")**print_info (is_readable ("/etc/passwd"), "readable")**print_info (is_writable ("/etc/passwd"), "writable")**print_info (is_modifiable ("/etc/passwd"), "readable and writable")**perm := permissions("/etc/passwd")***if** *perm.allow_group_read* **then***print ("Group is allowed to read /etc/passwd.%N")***else***print ("Group is not allowed to read /etc/passwd.%N")***end****if** *perm.allow_anyone_read_write* **then***print ("Anyone is allowed to read file.tmp.%N")***else***print ("Anyone is not allowed to read file.tmp.%N")***end****end***print_info (ok: BOOLEAN; what: STRING) is***do***print ("is_")**print (what)**print (" returned ")**print (ok)**print (".%N")***end****end**

Be aware that `POSIX_FILE_SYSTEM.is_readable` uses the real user and group IDs instead of the effective ones.

As can be seen in the above example, one can test for the permissions of a file using the `POSIX_PERMISSIONS` class. A new permissions class is created for every `POSIX_FILE_SYSTEM`.

`permissions` call, so it is best to cache this object. If the permissions change on the file system, this class does not reflect reality anymore, because it caches the permissions. Use `POSIX_PERMISSIONS.refresh` to update the contents. Use `set_allow_group_write`, `set_allow_anyone_read` and such to set permissions.

e-POSIX also gives you access to the `stat` function using the `POSIX_STATUS` class.

```
class EX_DIR4
```

```
inherit
```

```
    POSIX_FILE_SYSTEM
```

```
creation
```

```
    make
```

```
feature
```

```
    make is
```

```
        local
```

```
            stat: POSIX_STATUS
```

```
        do
```

```
            stat := status ("/etc/passwd")
```

```
            print ("size: ")
```

```
            print (stat.size.out)
```

```
            print (".%N")
```

```
            print ("uid: ")
```

```
            print (stat.permissions.uid)
```

```
            print (".%N")
```

```
        end
```

```
end
```

The `POSIX_STAT`, and through it `POSIX_PERMISSIONS`, are also returned by `POSIX_FILE_DESCRIPTOR.status`.

Browsing a directory can be done by allocated a `POSIX_DIRECTORY` class through the `POSIX_FILE_SYSTEM.browse_directory` feature:

```
class EX_DIR3
```

```
inherit
```

```
    POSIX_FILE_SYSTEM
```

```
creation
```

```
    make
```

```
feature
```

```

make is
  local
    dir: POSIX_DIRECTORY
  do
    from
      dir := browse_directory (".")
      dir.start
    until
      dir.exhausted
    loop
      print (dir.item)
      print ("%N")
      dir.forth
    end
    dir.close
  end
end

```

As can be seen, `POSIX_DIRECTORY` follows EiffelBase conventions.

When browsing a directory, all entries in that directory are returned. You might want to be interested only in certain files. e-POSIX has the ability to define arbitrary filters. Standard e-POSIX comes with an extension filter that only shows files with a certain extension:

```

class EX_DIR6

  inherit

    POSIX_FILE_SYSTEM

  creation

    make

  feature

    make is
      local
        dir: POSIX_DIRECTORY
      do
        from
          dir := browse_directory (".")
          dir.set_extension_filter (".e")
          dir.start
        until
          dir.exhausted
        loop
          print (dir.item)

```

```
        print ("%N")
        dir.forth
    end
    dir.close
end
end
```

In this chapter:

9.1 Introduction
9.2 Executing a child command
9.3 Catching a signal with Standard C
9.4 Catching a signal with POSIX
9.5 General wait for child handler
9.6 Forking a child process

9

Working with processes

9.1 Introduction

This chapter discusses starting processes, either by executing new ones or forking the current one. It also describes support for process communication using signals.

9.2 Executing a child command

Any command line can be executed by using the `POSIX_SHELL_COMMAND` class. Just pass a command line and `execute` it.

```
class EX_CMD
```

```
creation
```

```
make
```

```
feature
```

```
make is
local
  command: POSIX_SHELL_COMMAND
do
  create command.make ("/bin/ls *")
  command.execute
  print ("Exit code: ")
  print (command.exit_code)
  print ("%N")
end
```

```
end
```

Often one wants to redirect the output of the program that is being executed. For such cases use the `POSIX_EXEC_PROCESS` class.

```
class EX_EXECI
```

```
inherit
```

POSIX_CURRENT_PROCESS

creation

make

feature

```

make is
local
  ls: POSIX_EXEC_PROCESS
do
  -- list contents of current directory
  create ls.make_capture_output ("ls", <<"-l", ".>>)
  ls.execute
  print ("ls pid: ")
  print (ls.pid)
  print ("%N")
from
  ls.stdout.read_string (512)
until
  ls.stdout.eof
loop
  print (ls.stdout.last_string)
  ls.stdout.read_string (512)
end

-- close captured io
ls.stdout.close

-- wait for process
ls.wait_for (True)
end

```

end

Besides capturing output, you can also capture the standard input and standard error of the executed process.

It is important to wait for the child that has been executed at some point in time, just like any POSIX would have to do. If you do not wait for a child process, memory in the kernel is not released and eventually you would run out of processes. Also, after the `POSIX_EXEC_PROCESS.wait_for` command, the exit code of the process becomes available.

9.3 Catching a signal with Standard C

You can catch signals with Standard C. The following example demonstrates a program that can be safely interrupted by pressing Ctrl+C:

```

class EX_SIGNAL3

inherit

    EPX_CURRENT_PROCESS

    STDC_CONSTANTS

    STDC_SIGNAL_HANDLER

creation

    make

feature

    handled: BOOLEAN

    make is
    local
        signal: STDC_SIGNAL
    do
        create signal.make (SIGINT)
        signal.set_handler (Current)
        signal.apply

        print ("Wait 10s or press Ctrl+C.%N")
        sleep (10)
        if handled then
            print ("Ctrl+C pressed.%N")
        else
            print ("Ctrl+C not pressed.%N")
        end
    end

    signalled (signal_value: INTEGER) is
    do
        handled := True
    end

end

```

As Standard C doesn't have a sleep command, this program uses `EPX_CURRENT_PROCESS` to get either the `sleep` from POSIX or from Windows.

More explanation about the program itself can be found in [section 9.4](#).

9.4 Catching a signal with POSIX

Every class can become a signal handler by inheriting from `POSIX_SIGNAL_HANDLER`. Implement the `signalled` method as that is the function that is called when the signal occurs. Use `POSIX_SIGNAL.set_handler` to make your class a signal handler and call `apply` to start receiving signals when they occur.

The following examples demonstrates a program that can be safely interrupted by pressing Ctrl+C:

```
class EX_SIGNALI

inherit

    POSIX_CURRENT_PROCESS

    POSIX_CONSTANTS

    POSIX_SIGNAL_HANDLER

creation

    make

feature

    handled: BOOLEAN

    make is
    local
        signal: POSIX_SIGNAL
    do
        create signal.make (SIGINT)
        signal.set_handler (Current)
        signal.apply

        print ("Wait 30s or press Ctrl+C.%N")
        sleep (30)
        if handled then
            print ("Ctrl+C pressed.%N")
        else
            print ("Ctrl+C not pressed.%N")
        end
    end

    signalled (signal_value: INTEGER) is
    do
        handled := True
    end
```

end

All precautions and warnings when handling signals in C apply equally well in Eiffel of course. While in a signal handler, the signal will not be delivered again. Call `STDC_SIGNAL_HANDLER.reestablish` to make your signal handler interruptable.

You can write a single signal handler, that handles multiple signals. This makes it possible to have signal handling code in just one place. Create a class that inherits from `POSIX_SIGNAL_HANDLER`. Pass this class to the `POSIX_SIGNAL.set_handler` for every signal you want to catch. The signal value is passed as parameter to `POSIX_SIGNAL_HANDLER.signalled`, so you can write an **inspect** statement based on the value.

9.5 General wait for child handler

If you do not want to wait for every child process explicitly, you can write a simple SIGCHLD handler that just does a wait (I found this idea in (Stevens, 1998)):

class *EX_SIGNAL2*

inherit

POSIX_CURRENT_PROCESS

POSIX_CONSTANTS

POSIX_SIGNAL_HANDLER

creation

make

feature

make is

local

signal: POSIX_SIGNAL

do

create *signal.make (SIGCHLD)*

signal.set_handler (Current)

signal.apply

-- spawn child processes here

-- you dont have to wait for them

end

signalled (signal_value: INTEGER) is

do

wait

end

end

In Unix 98 you should be able to set the ignore handler for this signal. In pure POSIX systems the behaviour of the ignore handler is unspecified.

9.6 Forking a child process

Forking is very easy with this Eiffel POSIX implementation. The steps:

1. Write a child by inheriting from `POSIX_FORK_ROOT` and implementing its `execute` method.
2. The class that will do the forking, should inherit from `POSIX_CURRENT_PROCESS`.
3. Pass the child to the inherited feature `POSIX_CURRENT_PROCESS.fork` and the forking has begun.

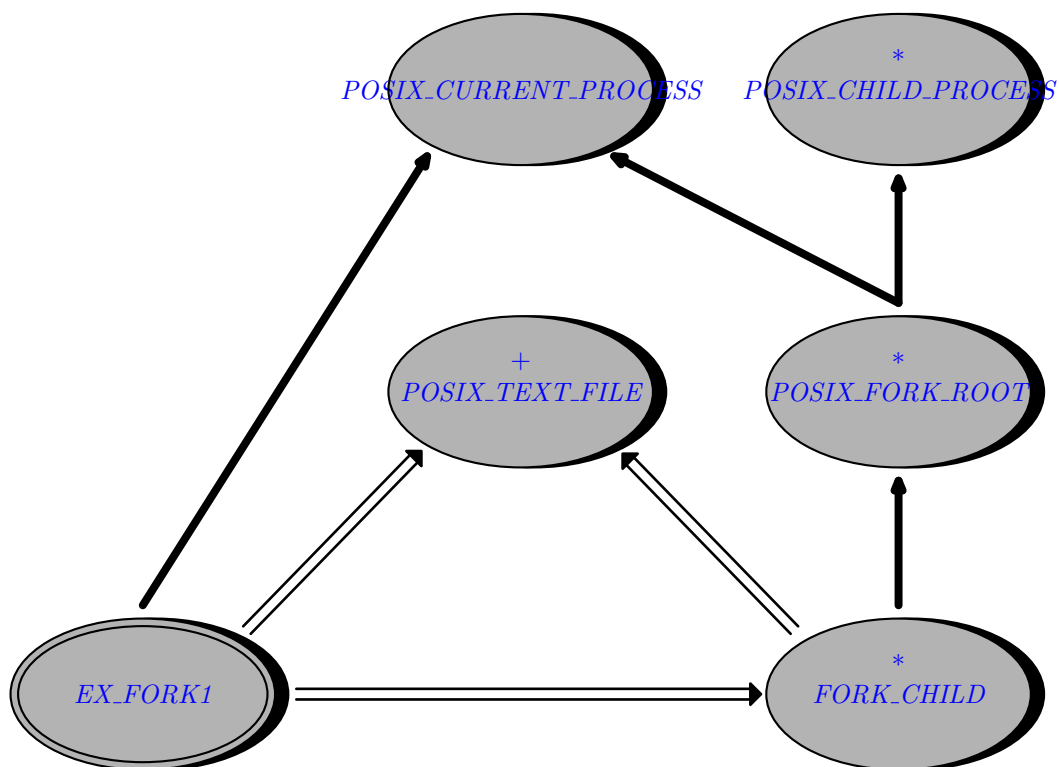


Figure 9.1 BON diagram of forking a child process.

The following class shows the process that forks the child.

class

EX_FORK1

inherit

POSIX_CURRENT_PROCESS

POSIX_FILE_SYSTEM

creation

make

feature

make is

local

reader: POSIX_TEXT_FILE

stop_sign: BOOLEAN

child: FORK_CHILD

do

-- necessary for SmallEiffel before -0.75 beta 7

ignore_child_stop_signal

unlink ("berend.tmp")

create_fifo ("berend.tmp", S_IRUSR + S_IWUSR)

create *child*

fork (child)

-- we will now block until file is opened for writing

create *reader.open_read ("berend.tmp")*

from

stop_sign := False

until

stop_sign

loop

reader.read_string (128)

print (reader.last_string)

stop_sign := equal(reader.last_string, "stop%N")

end

reader.close

-- now wait for the writer to terminate

child.wait_for (True)

unlink ("berend.tmp")

end

end

This class just displays anything that the writer, the child class, writes to the FIFO. When it recognizes stop, the reader stops after waiting for the child it has spawned. Note that this is very important! Wait for any child you have spawned else you might get spurious errors if the process exits and a child has not yet finished.

The following class shows the forked child.

```
class FORK_CHILD

inherit

    POSIX_FORK_ROOT

feature

    execute is
        local
            writer: POSIX_TEXT_FILE
        do
            create writer.open_append ("berend.tmp")
            writer.put_string ("first%N")
            writer.put_string ("stop%N")
            writer.close

            -- we give the reader some time to process these messages
            sleep (10)
        end
    end

end
```

In this chapter:

10 **Current time**

10 **Accessing environment variables**

10 **Capabilities**

10 **Querying the operating sys- tem**

10.1 Current time

e-POSIX has a very complete class to work with times. A time can be set from the current time by using `POSIX_TIME.make_from_now`. Before a time can be printed, it needs to be converted to either local time or UTC. Do this by calling `to_local` or `to_utc`. Date and times can be printed using features as `default_format`, `local_date_string`, `local_time_string` or a custom format through `format`.

class *EX_TIME1*

creation

make

feature

make is

local

time1,

time2: POSIX_TIME

do

create *time1.make_from_now*

time1.to_local

print_time (time1)

time1.to_utc

print_time (time1)

create *time2.make_time (0, 0, 0)*

print_time (time2)

create *time2.make_date_time (1970, 10, 31, 6, 55, 0)*

time2.to_utc

print_time (time2)

if *time2 < time1* **then**

print ("time2 is less than time1 as expected.%N")

else

print ("!! time2 is not less than time1.%N")

```
end
end

print_time (time: POSIX_TIME) is
do
  print ("Date: ")
  print (time.year)
  print ("-")
  print (time.month)
  print ("-")
  print (time.day)
  print (" ")
  print (time.hour)
  print (":")
  print (time.minute)
  print (":")
  print (time.second)
  print ("%N")
  print ("Weekday: ")
  print (time.weekday)
  print ("%N")
  print ("default string: ")
  print (time.default_format)
  print ("%N")
end

end
```

10.2 Accessing environment variables

Standard C supports reading environment variables with `STDC _ENV _VAR`.

```
class EX_ENV2

creation

  make

feature

  make is
  local
    env: STDC_ENV_VAR
  do
    create env.make ("HOME")
    print (env.value)
```

```

        print ("%N")
    end

```

```

end

```

The POSIX doesn't add any functionality here:

```

class EX_ENVI

```

```

creation

```

```

    make

```

```

feature

```

```

    make is
    local
        env: POSIX_ENV_VAR
    do
        create env.make ("HOME")
        print (env.value)
        print ("%N")
    end

```

```

end

```

It is not possible in POSIX to set an environment variable. This is possible with the Single Unix Specification classes. Using `SUS _ENV _VARset_value` it is possible to set environment variables.

10.3 Capabilities

Use the portable `EPX _SYSTEM` class to query for various system dependent constants like `max_open_files`. There are operating system dependent queries in `POSIX _SYSTEM` and `WINDOWS _SYSTEM`.

In this chapter:

11Sockets

11Echo client

11Echo client and server

11

Working with the network

11.1 Sockets

e-POSIX currently has fairly complete socket support. Not every option offered by the Single Unix Specification is supported yet, but as always we will attempt in every release to reach full support for every function offered.

As usual the EPX_XXXX classes are available on both Unix and Windows platform. The SUS_XXXX classes are available only on Single Unix Specification () systems and extend the EPX_XXXX classes with Unix specific functionality.

TCP functionality is available for both Windows and Unix. UDP is only available on Unix, as well as Unix streams.

11.2 Echo client

The following example demonstrates a simple echo client for TCP. An echo server must be running on your machine:

```
class EX_ECHO_CLIENT_TCP
```

```
creation
```

```
    make
```

```
feature
```

```
    hello: STRING is "Hello World.%N"
```

```
    make is
```

```
    local
```

```
        host: EPX_HOST
```

```
        service: EPX_SERVICE
```

```
        echo: EPX_TCP_CLIENT_SOCKET
```

```
        sa: EPX_HOST_PORT
```

```
    do
```

```
        create host.make_from_name ("localhost")
```

```
        create service.make_from_name ("echo", "tcp")
```

```
        create sa.make (host, service)
```

```

create echo.open_by_address (sa)
echo.put_string (hello)
echo.read_string (256)
if not echo.last_string.is_equal (hello) then
  print ("!! got: ")
  print (echo.last_string)
end
end

```

end

The following example demonstrates a simple echo client for UDP. An echo server must be running on your machine:

```

class EX_ECHO_CLIENT_UDP

creation

  make

feature

  hello: STRING is "Hello World.%N"

  make is
  local
    host: SUS_HOST
    service: SUS_SERVICE
    echo: SUS_UDP_CLIENT_SOCKET
    sa: EPX_HOST_PORT
  do
    create host.make_from_name ("localhost")
    create service.make_from_name ("echo", "udp")

    create sa.make (host, service)

    create echo.open_by_address (sa)
    echo.put_string (hello)
    echo.read_string (256)
    if not echo.last_string.is_equal (hello) then
      print ("!! got: ")
      print (echo.last_string)
    end
  end

end

```


11.3 *Echo client and server*

The following class demonstrates an echo server and client in a single class. It uses unix sockets (a fast interprocess communication) to achieve that.

```
class EX_ECHO_UNIX
```

```
inherit
```

```
    SUS_FILE_SYSTEM
```

```
    SUS_CONSTANTS
```

```
creation
```

```
    make
```

```
feature
```

```
    make is
```

```
        -- Echo client and server, unix style.
```

```
    local
```

```
        client_socket: SUS_UNIX_CLIENT_SOCKET
```

```
        server_socket: SUS_UNIX_SERVER_SOCKET
```

```
        client_fd: SUS_UNIX_SOCKET
```

```
        correct: BOOLEAN
```

```
    do
```

```
        if is_existing ("/tmp/eposix") then
```

```
            unlink ("/tmp/eposix")
```

```
        end
```

```
        create server_socket.listen_by_path ("/tmp/eposix", SOCK_STREAM)
```

```
        create client_socket.open_by_path ("/tmp/eposix", SOCK_STREAM)
```

```
        client_fd := server_socket.accept
```

```
        client_socket.put_string (hello)
```

```
        client_fd.read_string (256)
```

```
        correct := client_fd.last_string.is_equal (hello)
```

```
        if not correct then
```

```
            print ("Oops.%N")
```

```
        end
```

```
        client_fd.put_string (berend)
```

```
        client_socket.read_string (256)
```

```
        correct := client_socket.last_string.is_equal (berend)
```

```
        if not correct then
```

```
            print ("Oops.%N")
```

```
        end
```

```
        client_socket.close
```

```
        client_fd.close
```

```

    server_socket.close
    unlink ("/tmp/eposix")
end

```

feature {NONE} -- Implementation

```

hello: STRING is "Hello World.%N"
berend: STRING is "hello berend.%N"

```

end

The following class is similar, but uses TCP.

class EX_ECHO_TCP

inherit

SUS_CONSTANTS

creation

make

feature

```

make is
  -- Echo client and server, tcp style.
local
  host: SUS_HOST
  service: SUS_SERVICE
  client_socket: SUS_TCP_CLIENT_SOCKET
  server_socket: SUS_TCP_SERVER_SOCKET
  sa: EPX_HOST_PORT
  client_fd: ABSTRACT_TCP_SOCKET
  correct: BOOLEAN
do
  create host.make_from_name ("localhost")
  create service.make_from_port (port, "tcp")
  create sa.make (host, service)
  create server_socket.listen_by_address (sa)
  create client_socket.open_by_address (sa)
  client_fd := server_socket.accept
  client_socket.put_string (hello)
  client_fd.read_string (256)
  correct := client_fd.last_string.is_equal (hello)
  if not correct then
    print ("Oops.%N")
  end

```

```
    client_fd.put_string (berend)
    client_socket.read_string (256)
    correct := client_socket.last_string.is_equal (berend)
    if not correct then
        print ("Oops.%N")
    end

    client_socket.close
    client_fd.close
    server_socket.close
end

feature {NONE} -- Implementation

port: INTEGER is 9877
    -- Thanks to W. Richard Stevens

hello: STRING is "Hello World.%N"
berend: STRING is "hello berend%N"

end
```

In this chapter:

*12***Introduction**
*12***FTP client**
*12***HTTP client**
*12***HTTP server**
*12***IMAP4 client**
*12***IRC client**
*12***SMTP client**

12 **Working with the network: advanced topics**

12.1 Introduction

In version 2.0 e-POSIX has introduced the first of a series of classes for writing common Internet clients and servers.

Many of these classes are a work in progress, and might not have the robustness desired for critical applications.

12.2 FTP client

The e-POSIX FTP client supports almost all FTP operations, but currently has a fairly basic interface. Read and write operations return a stream for example. Reading and writing files to the file system is left as an exercise for the reader.

The following example demonstrates reading a directory from an FTP server and receiving a file:

```
class EX_FTP1
```

```
creation
```

```
make
```

```
feature
```

```
make is
```

```
local
```

```
ftp: EPX_FTP_CLIENT
```

```
do
```

```
-- ftp://ftp.nlm.nih.gov/nlmdata/sample/serfile/serfilesamp2005.xml
```

```
create ftp.make_anonymous (server_name, "guest")
```

```
ftp.open
```

```
if ftp.is_positive_completion_reply then
```

```
ftp.change_directory (directory_name)
```

```
ftp.name_list
```

```
dump_data_connection (ftp.data_connection)
```

```

    ftp.read_reply
    ftp.retrieve (file_name)
    dump_data_connection (ftp.data_connection)
    ftp.read_reply
    ftp.quit
    ftp.close
  else
    print ("Connect fails.%N")
  end
end

dump_data_connection (stream: KI_CHARACTER_INPUT_STREAM) is
  -- Dump stream input.
  require
    stream_not_void: stream != Void
  do
    from
      stream.read_character
    until
      stream.end_of_input
    loop
      print (stream.last_character)
      stream.read_character
    end
    stream.close
  end

feature -- Access

  directory_name: STRING is "/pub/FreeBSD"

  file_name: STRING is "README.TXT"

  server_name: STRING is "ftp.freebsd.org"

end

```

EXP_FTP_CLIENT also supports creating (`make_directory`) or deleting directories (`remove_directory`), deleting (`remove_file`), renaming (`rename_to`), and uploading files (`store`).

12.3 HTTP client

The following example demonstrates retrieval of a file through HTTP using the EXP_HTTP_10_CLIENT class:

```

class EX_HTTP1

  creation

```

```
make
```

feature

```
url: STRING is "http://www.freebsd.org/index.html"
```

```
make is
```

```
local
```

```
uri: EPX_URI
```

```
client: EPX_HTTP_10_CLIENT
```

```
do
```

```
create uri.make (url)
```

```
create client.make (uri.authority) -- www.freebsd.org
```

```
client.get (uri.path) -- /index.html
```

```
client.read_response
```

```
print (client.body.as_string)
```

```
end
```

```
end
```

It also demonstrates the use of the `EPX_URI` class to parse an URI into its components.

12.4 HTTP server

e-POSIX offers a basic HTTP server in `EPX_HTTP_SERVER`. You can create and register servlets which can respond to requests. A servlet is like a built-in CGI program. The server can also return files from the file system.

12.5 IMAP4 client

e-POSIX implements an IMAP4 client that supports IMAP4 access. The following example connects to an IMAP4 server and performs various operations:

```
class EX_IMAP4I
```

```
inherit
```

```
POSIX_CURRENT_PROCESS
```

```
creation
```

```
make
```

feature

```
make is
```

```
local
```

```

    client: EPX_IMAP4_CLIENT
do
  create client.make (host)
  if client.is_open then
    client.login (login_name, password)
    if client.response.is_ok then
      client.list_subscribed
      client.examine ("INBOX")
      client.fetch_message (4)
      print (client.response.current_message.message)
      client.close_mailbox
      client.logout
    else
      print ("Login failed.%N")
    end
    client.close
  else
    print ("Cannot connect to server.%N")
  end
end

feature -- Access

  host: STRING is "bmach"

  password: STRING is
  local
    password_env: STDC_ENV_VAR
  once
    create password_env.make ("IMAP4_PASSWORD")
    Result := password_env.value
  ensure
    password_not_void: Result /= Void
  end

end

```

The first operation is reading the list of available folders.. Next it examines the standard INBOX folder, i.e. open it for reading only. It reads message 4 and prints it. And finally it closes the mailbox.

The e-POSIX IMAP4 is fairly full featured, it can read and write messages and receive various pieces of information about the email such as just its header or its size.

12.6 IRC client

e-POSIX also has an IRC client implementation in `IRC_CLIENT`. Look at the test class `TEST_IRC_CLIENT` for an example, or download the Eiffel Bot from the eposix page.

12.7 SMTP client

`EPX_SMTP_CLIENT` implements support for sending email to an SMTP server. It only supports servers that can receive 8 bit messages. This class cannot convert 8 bit data to 7 bit data.

The following example demonstrates sending an email with this class:

```
class EX_SMTP1
```

```
creation
```

```
make
```

```
feature
```

```
make is
```

```
local
```

```
smtp: EPX_SMTP_CLIENT
```

```
message: EPX_MIME_EMAIL
```

```
mail: EPX_SMTP_MAIL
```

```
do
```

```
create smtp.make (smtp_server_name)
```

```
smtp.open
```

```
smtp.ehlo (my_domain)
```

```
create message.make
```

```
message.create_singlepart_body
```

```
message.header.set_from ("Berend de Boer", "berend@pobox.com")
```

```
message.header.set_to ("Berend de Boer", "berend@pobox.com")
```

```
message.header.set_subject ("EX_SMTP1")
```

```
message.text_body.append_string ("Hello!")
```

```
create mail.make (sender_mailbox, recipient_mailbox, message)
```

```
smtp.mail (mail)
```

```
smtp.quit
```

```
smtp.close
```

```
end
```

```
my_domain: STRING is "nederware.nl"
```

```
smtp_server_name: STRING is "localhost"
```

```
sender_mailbox: STRING is "berend"
```

```
recipient_mailbox: STRING is "berend"
```

```
end
```

The `EPX_SMTP_CLIENT`.`ehlo` command identifies the client with the server. Pass as argument the local domain, or if this is not available, the ip address of the client. The actual message is send after calling the `mail` command. It's argument is an `EPX_SMTP_MAIL` instance. This class is a

container for the sender, the recipients and the actual message that is to be sent. The message itself can be created by using an `EPX_MIME_EMAIL`. This class creates a MIME message, and has several convenience routines to quickly create such a message.

After the message has been sent, `EPX_SMTP_CLIENT.quit` is called to end the session and `close` is called to close the connection with the SMTP server.

The creation routine of `[EPX_SMTP_CLIENT]` takes as argument the SMTP server. Correctly finding the SMTP server for a given recipient involves querying a DNS server for MX records. However, passing the local SMTP server is usually sufficient as this server knows how to figure this out.

In this chapter:

13.1 Introduction

13.2 Windows

13.3 Creating a daemon

13.4 Logging messages and errors

13.5 SLM based logging

13

Writing daemons

13.1 Introduction

e-POSIX has several classes that help with writing daemons or services. First of all there is the `POSIX_DAEMON` ancestor class. But as daemons have no user interface, there are also classes for error and information logging.

13.2 Windows

On Windows NT (and derivatives) the equivalent of unix daemons are called services. They are a lot harder to write and require an Eiffel compiler with multi-threading. It is not yet possible to write an NT service with e-POSIX.

The logging functionality described in this chapter does work on Windows NT though.

13.3 Creating a daemon

Creating a simple daemon is easy if you inherit from `POSIX_DAEMON`. Implement the `execute` method, and you're done. At run-time, call `detach` to fork off a child. You can call `detach` as many times as you want to spawn daemons.

```
class EX_DAEMON
```

```
inherit
```

```
    POSIX_DAEMON
```

```
    ARGUMENTS
```

```
creation
```

```
    make
```

```
feature -- the parent
```

```
    make is
```

```
    do
```

```
        -- necessary under SmallEiffel
```

```
        ignore_child_stop_signal
```

```

if argument_count = 0 then
    print ("Options:%N")
    print ("-d    start daemon%N")
else
    if equal(argument(1), "-d") then
        detach
        print ("Daemon started.%N")
        print ("Its pid: ")
        print (last_child_pid)
        print ("%N")
    end
end
end

feature -- the daemon

    execute is
    do
        -- daemon stays alive for 20 seconds
        sleep (20)
    end

end

```

13.4 Logging messages and errors

Although POSIX doesn't have logging facilities, the Single Unix Specification does. This specification requires the presence of the `syslogd` daemon for centralizes logging facilities. The following example shows you to write messages to this daemon

```

class EX_SYSLOG

inherit

    SUS_CONSTANTS

    SUS_SYSLOG_ACCESSOR

creation

    make

feature

    make is
    do
        syslog.open ("test", LOG_ODELAY + LOG_PID, LOG_USER)
    end

```

```

        syslog.debug_dump ("this is a debug message")
        syslog.info ("this is an informational message")
        syslog.warning ("this is a warning")
        syslog.error ("this is an error message")

        syslog.close
    end

end

```

Always use the `SUS_SYSLOG_ACCESSOR` to access the syslog wrapper class `SUS_SYSLOG`. `SUS_SYSLOG` is a singleton, it makes no sense to open a connection to the syslog daemon twice.

13.5 ULM based logging

e-POSIX has portable routines for logging in Windows NT and Unix. This is build using the ULM (Universal Format for Logger Messages) specification. The specification itself can be found at <http://www.hsc.fr/gul/draft-abela-uml-05.txt>. It is a fixed format for logging that makes it easier to extract data with other tools.

On Unix e-POSIX outputs messages to the syslog daemon, see [section 13.4](#). On Windows e-POSIX logs to the event log. This makes this kind of logging specific to Windows NT based systems. It will not work on Windows 9x based systems.

Below a short example of using ULM. The first step is to create a handler that does the actual logging. The class `EPX_LOG_HANDLER` is operating system specific. If you compile on Windows it gives NT event log logging, on Unix it gives syslog logging. There is no logging mechanism for Windows 9x, but it should not be hard to write one. Just implement `ULM_LOG_HANDLER` and implement the deferred routines.

The second step is connecting that handler to the class that does ULM logging, the `ULM_LOGGING` class. Logging is now set up.

```

class EX_ULM

creation

make

feature -- Initialization

make is
local
    logger: ULM_LOGGING
    handler: EPX_LOG_HANDLER
    field: ULM_FIELD
    fields: ARRAY [ULM_FIELD]
do
    -- Create handler and logger

```

```

create handler.make (identification)
create logger.make (handler, system_name)

-- Log a simple message
logger.log_message (logger.Alert, subsystem_name, "Hello World.")

-- Log a message with a custom field
create fields.make (0, 0)
create field.make (logger.SRC_IP, "127.0.0.1")
fields.put (field, 0)
logger.log_event (logger.Usage, Void, fields)
end

feature -- Access

identification: STRING is "example"

system_name: STRING is "ex_ulm"

subsystem_name: STRING is "none"

end

```

Two messages are written. Below the slightly formatted output Unix:

```

Jul 21 21:12:34 dellius example: DATE=20030721091234 \
  HOST=dellius.nederware.nl PROG="ex_ulm.none" LVL=Alert \
  MSG="Hello World."
Jul 21 21:12:34 dellius example: DATE=20030721091234 \
  HOST=dellius.nederware.nl PROG="ex_ulm" LVL=Usage \
  SRC.IP=127.0.0.1

```

The first message is in the default format. This will always log the date, the host where the message originated and the program. The program field, PROG, consists of a system and subsystem name, separated by dots. This subsystem name is the second parameter to `ULM_LOGGING.log_message`. It may be Void, in which case no subsystem is added to the system name. The level field, LVL, contains the importance of the message. It is the first parameter to `ULM_LOGGING.log_message`. The class `ULM_LOG_LEVELS` has the complete list of levels. And in most cases the log ends with a simple message, MSG, that contains the message itself.

Feature `ULM_LOGGING.log_event` allows more control over the fields that are logged. That is demonstrated in the second message. You can pass the fields that are logged. You can use the fields listed in <http://www.hsc.fr/gul/draft-abela-ulm-05.txt>, or any other field. There is no MSG field if you don't specify one.

An interesting application of the ULM specification is the NetLogger library, see <http://www-didc.lbl.gov/NetLogger/>. It is a protocol to measure response times for a distributed application.

On Windows NT you can use the supplied `messages.dll` file to avoid this message in the event log:

The description for Event ID (some_number4) in Source (some_name) cannot be found. The local computer may not have the necessary registry information or message DLL files to display messages from a remote computer.

Register this DLL under the HKLM/SYSTEM/CurrentControlSet/Services/Eventlog/Application key. Add a new key which should have the name you have supplied to the [EPX_LOG_HANDLER.make](#) routine. This key should have two values:

1. EventMessageFile, type REG_SZ. Its value is the full path to this messages.dll file.
2. TypesSupported, type DWORD. Its value should be 7.

In this chapter:

14

Writing CGI programs

Although writing a CGI program doesn't really belong to POSIX, they still are very common, so I decided to include a few classes to make this easier. And of course, they build upon the Standard C classes.

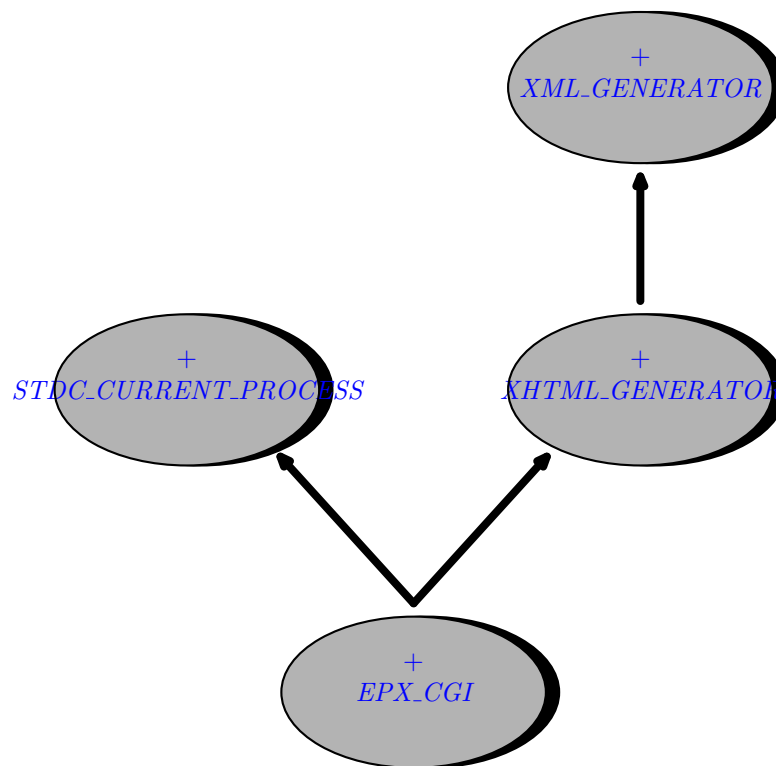


Figure 14.1 BON diagram of `EPX_CGI`.

You inherit from `EPX_CGI` and implement `execute`. As `EPX_CGI` itself inherits from `EPX_XHTML_WRITER` you can call use the features of that class to generate XHTML.

```
class EX_CGI
```

```
inherit
```

```
    EPX_CGI
```

```
creation
```

```
make

feature

    execute is
    do
        content_text_html

        doctype
        b_html

        b_head
        title ("e-POSIX CGI example.")
        e_head

        b_body

        p ("Hello World.")
        extend ("<p>you can use your <b>own</b> tags.</p>")
        b_p
        puts ("or use any tag by using:")
        e_p

        start_tag ("table")
        set_attribute ("border", Void)
        set_attribute ("cols", "3")
        start_tag ("tr")
        start_tag ("td")
        add_data ("start_tag")
        stop_tag
        start_tag ("td")
        add_data ("stop_tag")
        stop_tag
        stop_tag
        stop_tag

        e_body
        e_html

    end

end
```

Output is accumulated in a string and written to stdout after your `EPX_CGI.execute` method has finished. The partially built string is accessible with `EPX_XML_WRITER.unfinished_xml`. Generated output is XHTML, which usually displays fine with older browsers. If strict XHTML is problematic, you can call `doctype_transitional` instead of `doctype`.

It is important not to write to stdout as the output is only written after your `EPX_CGI.execute` has finished. If you want to write something to standard output, use the `EPX_CGI.add_data` feature or its shortcut alias `puts`. If you want to write real tags, use `add_raw`. This last feature allows you to write anything, while `puts` escapes reserved characters like '>'.

If you use provided features like `b_a`, `b_p` and such, an attempt is made to produce good looking source. Also your input is somewhat validated against XHTML standards.

It is also easy to write a CGI program that displays a form and accepts submitted values. Even file upload is supported. The following example uses the GET method to submit data:

```
class EX_CGI2
```

```
inherit
```

```
EPX_CGI
```

```
creation
```

```
make
```

```
feature
```

```
execute is
```

```
do
```

```
content_text_html
```

```
doctype
```

```
b_html
```

```
b_head
```

```
title ("e-POSIX CGI form example.")
```

```
e_head
```

```
b_body
```

```
b_form_get ("ex_cgi2.bin")
```

```
b_p
```

```
puts ("Name: ")
```

```
b_input ("text", "name")
```

```
set_attribute ("size", "32")
```

```
e_input
```

```
e_p
```

```
b_p
```

```
puts ("City: ")
```

```
input_text ("city", 40, "enter city here")
```

```
e_p
```

```
b_p
b_button_submit ("action", "GO!")
e_button_submit

nbsp

button_reset
e_p

e_form

hr

p ("In your last submit you entered:")
b_p
if not has_key ("name") then
    puts ("!!!!")
end
puts ("name: ")
puts (value ("name"))
puts (" ")
puts ("city: ")
puts (raw_value ("city"))
e_p

e_body
e_html

end
```

end

You can use `EPX_CGI.b_input` to start an input element as shown for the input of a name. Or you can use `input_text` to start a simple text input as shown for the input of a city. Below the line you see the value a user has submitted, if any. Use `value` to get values with certain meta-characters removed. The output is still not safe to be passed straight to a Unix Shell though! You can use `raw_value` to get the contents as submitted by the user.

In the above example it doesn't matter much if you use `b_form_get` or `b_form_post`. But with the GET method, you cannot upload files. The following example demonstrates how files can be uploaded:

```
class EX_CGI3
```

```
inherit
```

```
EPX_CGI
```

```
creation
```

make

feature

execute is

do

content_text_html

assert_key_value_pairs_created

save_uploaded_files

doctype

b_html

b_head

title ("e-POSIX CGI file upload example.")

e_head

b_body

b_form ("post", "ex_cgi3.bin")

set_attribute ("enctype", mime_type_multipart_form_data)

b_p

puts ("Filename: ")

b_input ("file", "filename")

set_attribute ("size", "32")

set_attribute ("maxlength", "128")

e_input

e_p

b_p

b_button_submit ("action", "Upload file(s)")

e_button_submit

nbsp

button_reset

e_p

e_form

e_body

e_html

end

```
save_uploaded_files is
local
  kv: EPX_KEY_VALUE
  buffer: STDC_BUFFER
  target_name: STRING
  target: STDC_BINARY_FILE
do
  create buffer.allocate (8192)
  from
    cgi_data.start
  until
    cgi_data.after
  loop
    kv := cgi_data.item_for_iteration
    if kv.file /= Void then
      from
        target_name := "/tmp/" + kv.value
        create target.create_write (target_name)
        kv.file.read_buffer (buffer, 0, 8192)
      until
        kv.file.eof
      loop
        target.write_buffer (buffer, 0, kv.file.last_read)
        kv.file.read_buffer (buffer, 0, 8192)
      end
      target.close
      kv.file.close
    end
    cgi_data.forth
  end
  buffer.deallocate
end

end
```

It is important to set the encoding type. This example accepts a file and writes it to /tmp. Because multiple files can be present, this example just loops over all key value pairs and checks if a file is present. This example isn't fool-proof with multiple users submitting the same file, but you should get the idea.

Note that the first line is `EPX_CGI.content_text_html`: in case an exception occurs, the web server is still able to output something back to the user.

After that we make sure that the key value pairs are created with `assert_key_value_pairs_created`. They are automatically created if you call `value`, but in this case we want the key value pairs themselves. In `EX_CGI3.save_uploaded_files` we use the `EPX_KEYVALUE.file` feature to check if that key value pair is an uploaded file: if it is not `Void`, it points to a temporary file. As this file will be deleted when it is closed or when your program exits, we have

to copy it to a new file. The filename is just the value part of this key value pair. The filename is guaranteed to be free of directory parts.

In the last example we just print all key/value pairs to the file `list.txt` in the temporary directory. We redirect the user to another file.

```
class EX_CGI4

inherit

    EPX_CGI

    EPX_FACTORY

creation

    make

feature

    execute is
    do
        assert_key_value_pairs_created
        save_values

        extend ("Location: /mydir/myfile.html")
        new_line
        new_line
    end

    save_values is
    local
        fout: STDC_TEXT_FILE
        kv: EPX_KEY_VALUE
    do
        create fout.create_write (fs.temporary_directory + "/list.txt")
    from
        cgi_data.start
    until
        cgi_data.after
    loop
        kv := cgi_data.item_for_iteration
        fout.puts (kv.key)
        fout.puts ("%T")
        fout.puts (kv.value)
        fout.puts ("%N")
        cgi_data.forth
    end
```

```
fout.close  
end
```

```
end
```

In this chapter:

*15***Error handling with exceptions**

*15***Manual error handling**

15

Error handling

This chapter describes the error handling strategies that are possible with e-POSIX. Basically there are two strategies: using the Eiffel exception mechanism or doing the error handling all yourself.

15.1 Error handling with exceptions

The opinion of the author of e-POSIX is that Eiffel's exception mechanism is very well suited to deal with things like files that cannot be opened or directories that do not exist. Others disagree, see [section 15.2](#). e-POSIX is designed such that when a POSIX routine returns an error code, an exception is thrown. Here my arguments why I favor this style of error handling:

1. We all know that exceptions are to be used for breach of contract. This idea is formulated in (Meyer, 1997) and is the best expressed opinion of exception handling I know.
So if you ask an e-POSIX method to open a file, it will do that for you. If it cannot open the file, for whatever reason, it will raise an exception. The same argument holds if you ask it to go to a directory, to start a program, or to open a connection to another machine.
This approach is also reflected in the names of e-POSIX's features. The name is `POSIX_TEXT_FILE.open_read` and not `POSIX_TEXT_FILE.attempt_open_read`.
2. It is usually not wise to trust clients with error handling. The larger a distance between a software failure and the error report, the more difficult it is to make a correct diagnosis of what went wrong (see (Hatton, 2001)). e-POSIX uses the fail early, fail hard approach.
3. Error handling is often forgotten or left to some global general error handling mechanism. In an interesting article (see (Whittaker, 2001)) James Whittaker describes how he modified certain system calls to return legitimate, but unexpected return codes. Memory allocation failed for example, or opening a file returned with no more file handles. Applications failed within seconds, but it was usually completely unclear why.
4. It's a lot easier for programmer's. You don't have to write any error handling. If your program completed, you know that there wasn't a single system call that failed, that you didn't continue despite some error. This will make it possible to write programs that do their work correctly if no errors occur, or else do nothing.

First an example. Let's take a look at the code you have to write in case you want to handle failure of opening a file:

```
class EX_ERRORI
```

```
inherit
```

```
    POSIX_CURRENT_PROCESS
```

```
creation
```

```

make

feature

make is
  local
    fd: POSIX_FILE_DESCRIPTOR
  do
    fd := attempt_create_file
  end

attempt_create_file: POSIX_FILE_DESCRIPTOR is
  local
    attempt: INTEGER
    still_exists: BOOLEAN
  do
    create Result.create_with_mode ("myfile", O_CREAT+O_TRUNC+O_EXCL, 0)
  rescue
    still_exists := errno.value = EEXIST
    attempt := attempt + 1
    if still_exists and then attempt <= 3 then
      sleep (1)
      retry
    end
  end
end

end

```

In this example we try to create a file exclusively. The create will fail if the file already exists. In case this happens, we retry 3 times. Before retrying we wait 1 second. Note that if the error is not `EEXIST`, we fail directly, without retrying.

In my opinion above's code is just the code you want to write usually: do not worry about errors, if something goes wrong, your application will fail.

My preferred way of error handling is (or sometimes should be) also reflected in the preconditions. For example the `POSIX_FILE_SYSTEM.browse_directory` has the precondition that the given path should exist and should be a directory. Quite reasonable I think. The argument against such preconditions is that it is somewhat strange: if a client has honoured the precondition by checking that the directory exists, it should be able to assume that it safely can call the routine. But between its own check and the actual call, the directory can be removed by another process.

This is the concurrent precondition paradox (see (Meyer, 1997)). In my opinion it would not be wise to remove this precondition. It is true that honouring it, will not make sure the contract is not broken. But it still serves a very useful purpose: documentation.

For example the routine `POSIX_FILE_SYSTEM.remove_file` does not have the precondition that the file should exist. That isn't an oversight. This routine does not fail if the file no longer exists for good reason: it honours its postcondition after all. So when you call this routine, the file may or may not exist. The routine doesn't care.

15.2 Manual error handling

In spite of the arguments listed in the previous section, automatic error handling is perhaps tedious to use when you expect a lot of errors. And some programmers just do not like Eiffel's exception mechanism. Therefore e-POSIX implements a completely different style of error handling. In this case, e-POSIX continues when an error occurs, but it saves the errorcode, and you can check the errorcode of the first error when you wish. This first errorcode has to be reset by the programmer. An example:

```

class EX_ERROR2

  inherit

    STDC_SECURITY_ACCESSOR

  creation

    make

  feature

    make is
      local
        fd: POSIX_FILE_DESCRIPTOR
      do
        security.error_handling.disable_exceptions
        create fd.create_write ("myfile")
        if fd.errno.first_value = 0 then
          fd.put_string ("1%N")
          fd.put_string ("2%N")
          fd.close
        else
          fd.errno.clear_first
        end
      end
    end

  end
end

```

Exception handling is turned off by a call to `STDC_SECURITY_ACCESSOR`. `security.error_handling.disable_exceptions`. It can be enabled again by calling `security.error_handling.enable_exceptions`. In between, you're on your own, just like a C programmer. If `myfile` cannot be opened, nothing happens, and the `POSIX_FILE_DESCRIPTOR.put_string` feature is called. Depending if you have enabled precondition checking or not, `put_string` will fail. The precondition if `put_string` is that the file has to be open. Therefore, at certain points, you're still forced to deal with errors. Every object has an `errno` variable. This variable points to the global `STDC_ERRNO` object (its a once routine). So there basically is just one `first_value` error value. Whatever object caused the error, you can check the `errno.first_value` of any e-POSIX object. The last error is still available in `errno.value`.

If there is no error, the program continues writing. If `POSIX_FILE_DESCRIPTOR.put_string` failed, the next one is still executed. If there is an error, we reset it with `STDC_ERRNO.clear_first`. This gives us the chance to catch another error value if an error occurs. If this method is not called, `first_value` will keep its original value.

The following example is the same as `EX_ERROR1`. It shows how to open a file exclusively with manual error handling.

```
class EX_ERROR3
```

```
inherit
```

```
    POSIX_CURRENT_PROCESS
```

```
    EXCEPTIONS
```

```
creation
```

```
    make
```

```
feature
```

```
    make is
```

```
        local
```

```
            fd: POSIX_FILE_DESCRIPTOR
```

```
        do
```

```
            security.error_handling.disable_exceptions
```

```
            fd := attempt_create_file
```

```
        end
```

```
attempt_create_file: POSIX_FILE_DESCRIPTOR is
```

```
    require
```

```
        manual_error: not security.error_handling.exceptions_enabled
```

```
    local
```

```
        attempt: INTEGER
```

```
        still_exists: BOOLEAN
```

```
    do
```

```
        from
```

```
            attempt := 1
```

```
            still_exists := True
```

```
        until
```

```
            not still_exists or else attempt > 3
```

```
        loop
```

```
            create Result.create_with_mode ("myfile", O_CREAT+O_TRUNC+O_EXCL, 0)
```

```
            still_exists := errno.first_value = EEXIST
```

```
            if still_exists then
```

```
                sleep (1)
```

```
                attempt := attempt + 1
```

```
        end
    end
    if still_exists then
        raise ("failed to create file")
    end
end

end
```

As you can see, manual error handling does not necessarily translate into less code.

The summary of this section is that you should check each distinctive step when using manual error handling. You don't have to check intermediate steps.

In this chapter:

16 Denial of service attacks

16 Authorization bypass attacks

16

Security

e-POSIX is well-suited to write server applications like CGI scripts and daemons. As these applications can be hosted on servers that are attached to the Internet, they could be prone to attack. Applications written with e-POSIX could be misused in a denial of service attack or to gain root access. e-POSIX offers certain protection mechanisms that enable your applications to fend off such penetrations.

This chapter shows you how applications can be misused and what mechanisms e-POSIX offers for certain attacks.

“Programmers typically focus on “positive” aspects of programs, that is, what is the functionality required for the task to be accomplished. Programmers rarely focus on the negative aspects of programs, that is, what functionality is not required for the program to accomplish its task. Attackers take advantage of programmers failure to consider negative functionality. Perhaps a reason that programmers avoid negative functionality is that there is no good way to specify what a program should not be permitted to do.”

16.1 Denial of service attacks

In a denial of service attack, crackers attempt to deplete one or more finite resources. Resources can be software related like database connections or TCP/IP connections, but ultimately resources are finite because of hardware limitations. This manual distinguishes the following hardware resources:

- Memory.
- CPU.
- Disk space.
- Network bandwidth.

A denial of service attack succeeds if a cracker depletes these resources in such a way that the server cannot handle request anymore, or handles them very slowly. For example, Linux 2.2 is easy to bring to its knees if you keep on allocating memory. In normal situations your application runs fine, and allocates only a limited amount of memory. But an attacker might have found a way to make your application allocate much more memory. Even if you are sure that the code you have written is not prone to such an attack, you might use a library based on e-POSIX that does have code that is exploitable.

e-POSIX has some limited support to set limits on memory, file handle (a memory issue) and cpu usage. When a set limit has been exceeded, an exception is raised.

To limit the amount of memory that can be allocated by the `STDC_BUFFER` class, inherit from `STDC_SECURITY_ACCESSOR` and call `security.memory.set_max_allocation`. Currently this limits the amount of memory that can be allocated with `STDC_BUFFER`. It does not limit the amount of memory that is allocated by `STRING` or other classes. You can also limit

the amount of memory that can be allocated with a single call by calling `security.memory.set_max_single_allocation`.

You can limit the number of file handles a program can open by calling `security.files.set_max_open_files`. This works only with files and sockets opened by e-POSIX classes as `STDC_FILE` and `POSIX_FILE_DESCRIPTOR`, not with files opened through other means. In this case you cannot rely on the garbage collection to close your file. Certain garbage collectors do not allow calling other classes in the `MEMORY.dispose` method. e-POSIX needs to do this to decrement its idea of the number of open handles. Only when you explicitly call `STDC_FILE.close` will the e-POSIX decrease its open file handles.

You can limit the amount of CPU time by calling `security.cpu.set_max_process_time`. It is not possible to automatically halt your application when this time has exceeded. You have to call `security.cpu.check_process_time` to actually check the processor time used.

Currently e-POSIX cannot check disk space or network bandwidth limitations.

Discuss here that decrementing only works for manual deallocations, I'm very sorry about that, but this is a problem of ISE. I'm thinking about ways to work around this.

16.2 Authorization bypass attacks

A hacker can bypass authorization if he or she, through your program, can gain the following access:

- Access to more information than your program is written to provide. Security is not breached here, but your program is used in an 'innovative' way. Note that if your program runs within the root security context (suid root), security can be breached!
- Security is breached when your program is used to get more access rights than your program is written to provide. Especially suid root programs are an attractive target here.

Usually Eiffel programs do not allocate buffers on the stack, so they are not prone to the so called 'buffer overflow' attack. As certain vendors might provide some 'native' class that allocate things on the stack, leave precondition checking always on in suid root programs.

Currently e-POSIX doesn't offer much protection for suid root programs. Much better security will be the topic of a next release.

In this chapter:

17 Making C Headers available to Eiffel
17 Distinction between Standard C and POSIX headers
17 B translation details

17

Accessing C headers

This chapter explains the conventions that e-POSIX uses to access the C-headers.

17.1 Making C Headers available to Eiffel

The most portable and safest header translation comes when a C function is not called verbatim, but instead a translation function is used. For example to make the Standard C function `fopen` available within Eiffel a new header file is created which lists an Eiffel compatible way to call this routine:

```
#include "eiffel.h"
#include <stdio.h>
```

```
EIF_POINTER posix_fopen(EIF_POINTER filename, EIF_POINTER mode);
```

Instead of using C types, we use Eiffel types here, which are made available by including `eiffel.h`.

The corresponding C file contains the following implementation:

```
#include "my_new_header.h"

EIF_POINTER posix_fopen(EIF_POINTER filename, EIF_POINTER mode)
{
    return ( (EIF_POINTER) fopen (filename, mode));
}
```

It simply calls the original function, returning the result. Type conversion between Eiffel and C types shouldn't pose problems this way.

To be able to call this function from Eiffel, an **external** feature needs to be written. For example:

```
class HEADER_STDIO
```

```
feature {NONE} -- C binding for stream functions
```

```
    posix_fopen (path, a_mode: POINTER): POINTER is
```

```
        -- Opens a stream
```

```
    require
```

```
        valid_mode: a_mode /= default_pointer
```

```
    external "C"
```

```
    end
```

end

Of course, the Eiffel function can have all Design By Contract features Eiffel programmers are accustomed too.

To recapitulate: every header that is to be translated, needs:

1. a new header file, and
2. a corresponding C file, and
3. an Eiffel class.

For example to translate `<stdio.h>` a header file like `eiffel_stdio.h` and a C file `eiffel_stdio.c` is needed. The Eiffel class could be in `header_stdio.e`.

17.2 Distinction between Standard C and POSIX headers

However, POSIX sometimes defines extensions to existing Standard C headers. Simply using a translation header file like `eiffel_stdio.h` will not work for pure Standard C Eiffel programs, as it can include POSIX specific extensions that might simply not be available on a given platform.

Therefore, e-POSIX divides the C headers in several groups:

1. The Standard C headers.
2. The POSIX headers.
3. The Single Unix Specification headers.
4. Microsoft Windows headers (as far as they define POSIX functions, this library does not translate Microsoft Windows specific functions).

Every group gets its own translation header with its own prefix. A translated header has a prefix, an underscore and next the original header name. The Standard C translation of `<stdio.h>` is done in `c_stdio.h` and `c_stdio.c`. The POSIX extensions to this header are available in `p_stdio.h` and `p_stdio.c`.

The corresponding Eiffel class follows similar conventions. It has the group's prefix, next the string 'API', an underscore and next the name of the header. So all `<stdio.h>` functions are made available in `CAPI _STDIO`.

In [table 17.1](#) all the groups with there translation header prefix and Eiffel class prefix are listed. See also the directory structure in [figure 17.1](#).

17.3 C translation details

This translation wants to do as less as possible at the C level. It attempts to just make available the C constants and C functions and do the actual work in Eiffel.

A few details:

1. Constants, C macro definitions, are exported in the header file with the prefix 'const_' and next the macro name. The Eiffel API class exports these constants with the original, uppercased name.
2. Struct members are exported with getter and setter functions. The get function has the prefix 'posix', an underscore, the struct name, an underscore and as last the member name. The

set function has the prefix ‘posix’, an underscore, ‘set’, an underscore, the struct name, an underscore and as last the member name.

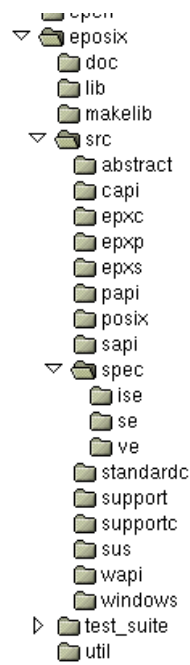


Figure 17.1 e-POSIX
directory structure

Group	directory	header prefix	class prefix
Standard C	src/capi	c	CAPI
POSIX	src/[api	p	PAPI
Single Unix Specification	src/sapi	s	SAPI
Windows	src/wapi	w	WAPI

Table 17.1 e-POSIX prefix conventions

In this chapter:

A

Posix function to Eiffel class mapping list

The following table defines exactly where a given Posix function is used in a Eiffel class mapping. The table is sorted in alphabetic order. Note that when a STDC_ class is listed, the feature is also available in the corresponding POSIX_ class. The same is true for the EPX_ classes. The EPX_ classes provide functionality portable between Unix and Windows. The corresponding POSIX_ or SUS_ classes extend that functionality for or the Single Unix Specification.

Function	Header	Class	
abort	<stdlib.h>	STDC_CURRENT_PROCESS.abort	
accept	<sys/socket.h>	EPX_TCP_SERVER_SOCKET.accept	
access	<unistd.h>	ABSTRACT_FILE_SYSTEM.is_accessible	
aio_cancel	<aio.h>	POSIX_ASYNC_IO_REQUEST.cancel	
aio_error	<aio.h>	POSIX_ASYNC_IO_REQUEST.is_pending	
aio_fsync	<aio.h>	POSIX_ASYNC_IO_REQUEST.synchronize	
aio_read	<aio.h>	POSIX_ASYNC_IO_REQUEST.read	
aio_return	<aio.h>	POSIX_ASYNC_IO_REQUEST.return_status	
aio_suspend	<aio.h>	POSIX_ASYNC_IO_REQUEST.wait_for	
aio_write	<aio.h>	POSIX_ASYNC_IO_REQUEST.write	
alarm	<unistd.h>	POSIX_TIMED_COMMAND	
asctime	<time.h>	STDC_TIME.default_format	
atexit	<stdlib.h>	STDC_EXIT_SWITCH.install	
bind	<sys/socket.h>	EPX_TCP_SERVER_SOCKET.listen_by_address	
calloc	<stdlib.h>	STDC_BUFFER.allocate_and_clear	
cfgetispeed	<termios.h>	POSIX_TERMIOS.input_speed	
cfgetospeed	<termios.h>	POSIX_TERMIOS.output_speed	
cfsetispeed	<termios.h>	POSIX_TERMIOS.set_input_speed	
cfsetospeed	<termios.h>	POSIX_TERMIOS.set_output_speed	
chdir	<unistd.h>	POSIX_FILE_SYSTEM.change_directory	
chmod	<sys/stat.h>	POSIX_FILE_SYSTEM.change_mode	
chown	<unistd.h>	POSIX_PERMISSIONS_PATH.apply_owner_and_group	
clearerr	<stdio.h>	STDC_FILE.clear_error	
clock	<time.h>	STDC_CURRENT_PROCESS.clock	
clock_getcpuclockid	<time.h>		
clock_getres	<time.h>	SUS_SYSTEM.real_time_clock_resolution	
clock_gettime	<time.h>	SUS_SYSTEM.real_time_clock	
clock_nanosleep	<time.h>		
clock_settime	<time.h>		
close	<unistd.h>	EPX_FILE_DESCRIPTOR.close	
closedir	<dirent.h>	POSIX_DIRECTORY	
closelog	<syslog.h>	SUS_SYSLOG.close	
confstr	<unistd.h>		
connect	<sys/socket.h>	EPX_TCP_CLIENT_SOCKET.open_by_address,open_by_name_and_port	
creat	<fcntl.h>	EPX_FILE_DESCRIPTOR.create_read_write	

ctermid	<unistd.h>	
ctime	<time.h>	
cuserid	<stdio.h>	
daylight	<time.h>	
difftime	<time.h>	STDC_TIME
dup	<unistd.h>	EPX_FILE_DESCRIPTOR.make_as_duplicate
dup2	<unistd.h>	EPX_FILE_DESCRIPTOR.make_as_duplicate
endgrent	<grp.h>	
endhostent	<netdb.h>	
endnetent	<netdb.h>	
endprotoent	<netdb.h>	
endpwent	<pwd.h>	
endservent	<netdb.h>	
execl	<unistd.h>	
execle	<unistd.h>	
execlp	<unistd.h>	
execv	<unistd.h>	
execve	<unistd.h>	
execvp	<unistd.h>	EPX_EXEC_PROCESS.execute
exit	<stdlib.h>	STDC_CURRENT_PROCESS.exit
_exit	<unistd.h>	
fchmod	<sys/stat.h>	
fchown	<sys/stat.h>	
fclose	<stdio.h>	STDC_FILE.close
fcntl	<unistd.h>	POSIX_FILE_DESCRIPTOR
fdatasync	<unistd.h>	POSIX_FILE_DESCRIPTOR.synchronize_data
fdopen	<stdio.h>	POSIX_FILE.make_from_file_descriptor
feof	<stdio.h>	STDC_FILE.eof
ferror	<stdio.h>	STDC_FILE.error
fflush	<stdio.h>	STDC_FILE.flush
fgetc	<stdio.h>	STDC_FILE.get_character
fgetpos	<stdio.h>	STDC_FILE.get_position
fgets	<stdio.h>	STDC_FILE.get_string
fileno	<stdio.h>	POSIX_FILE_DESCRIPTOR.make_from_file
flockfile	<stdio.h>	
fopen	<stdio.h>	STDC_FILE
fork	<unistd.h>	POSIX_CURRENT_PROCESS.fork
fpathconf	<unistd.h>	
fprintf	<stdio.h>	
fputc	<stdio.h>	STDC_FILE.putc
fputs	<stdio.h>	STDC_FILE.put_string
fread	<stdio.h>	STDC_FILE.read
free	<stdlib.h>	STDC_BUFFER.deallocate
freopen	<stdio.h>	STDC_FILE.reopen
fseek	<stdio.h>	STDC_FILE.seek
fsetpos	<stdio.h>	STDC_FILE.set_position
fstat	<sys/stat.h>	POSIX_STATUS
fsync	<unistd.h>	POSIX_FILE_DESCRIPTOR.synchronize
ftell	<stdio.h>	STDC_FILE.tell

ftruncate	<unistd.h>	
ftrylockfile	<stdio.h>	
funlockfile	<stdio.h>	
fwrite	<stdio.h>	STDC_FILE.write
getc	<stdio.h>	
getchar	<stdio.h>	
getcwd	<unistd.h>	POSIX_FILE_SYSTEM.current_directory
getegid	<unistd.h>	POSIX_CURRENT_PROCESS.effective_group_id
getenv	<stdlib.h>	STDC_ENV_VAR.value
geteuid	<unistd.h>	POSIX_CURRENT_PROCESS.effective_user_id
getgid	<unistd.h>	POSIX_CURRENT_PROCESS.real_group_id
getgrgid	<grp.h>	POSIX_GROUP.make_from_gid
getgrnam	<grp.h>	POSIX_GROUP.make_from_name
getgroups	<unistd.h>	POSIX_CURRENT_PROCESS.is_in_group
getlogin	<unistd.h>	POSIX_CURRENT_PROCESS.login_name
getpgrp	<unistd.h>	POSIX_CURRENT_PROCESS.process_group_id
getpid	<unistd.h>	POSIX_CURRENT_PROCESS.pid
getppid	<unistd.h>	POSIX_CURRENT_PROCESS.parent_pid
getpwnam	<pwd.h>	POSIX_USER.make_from_name
getpwuid	<pwd.h>	POSIX_USER.make_from_uid
gets	<stdio.h>	
gettimeofday	<sys/time.h>	
getuid	<unistd.h>	POSIX_CURRENT_PROCESS.real_user_id
gmtime	<time.h>	STDC_TIME.to_utc
inet_ntoa	<arpa/inet.h>	EPX_IP4_ADDRESS.out
isatty	<unistd.h>	EPX_FILE_DESCRIPTOR.is_attached_to_terminal
htonl	<netinet/in.h>	SAPI_IN.posix_htonl
htons	<netinet/in.h>	SAPI_IN.posix_htons
kill	<signal.h>	POSIX_PROCESS.kill
link	<unistd.h>	POSIX_FILE_SYSTEM.link
lio_listio	<aio.h>	
localeconv	<locale.h>	STDC_LOCALE_NUMERIC
localtime	<time.h>	STDC_TIME.to_local
lseek	<unistd.h>	EPX_FILE_DESCRIPTOR.seek
malloc	<stdlib.h>	STDC_BUFFER.allocate
memcpy	<string.h>	STDC_BUFFER.memory_copy
memchr	<string.h>	
memcmp	<string.h>	CAPI_STRING.posix_memcmp
memmove	<string.h>	STDC_BUFFER.memory_move
memset	<string.h>	STDC_BUFFER.fill_with
mkdir	<sys/stat.h>	POSIX_FILE_SYSTEM.make_directory
mkfifo	<sys/stat.h>	POSIX_FILE_SYSTEM.create_fifo
mktime	<time.h>	STDC_TIME.set_date_time
mlockall	<sys/mman.h>	
mlock	<sys/mman.h>	
mmap	<sys/mman.h>	POSIX_MEMORY_MAP
mprotect	<sys/mman.h>	
mq_receive	<mqqueue.h>	
mq_close	<mqqueue.h>	
mq_getattr	<mqqueue.h>	
mq_notify	<mqqueue.h>	
mq_open	<mqqueue.h>	
mq_send	<mqqueue.h>	
mq_setattr	<mqqueue.h>	
mq_unlink	<mqqueue.h>	
msync	<sys/mman.h>	
munlockall	<sys/mman.h>	

munlock	<sys/mman.h>	POSIX_MEMORY_MAP
munmap	<sys/mman.h>	Efeature[SUS_CURRENT_PROCESS]nanosleep
nanosleep	<time.h>	SAPI_IN.posix_ntohl
ntohl	<netinet/in.h>	SAPI_IN.posix_ntohs
ntohs	<netinet/in.h>	EPX_FILE_DESCRIPTOR.open
open	<fcntl.h>	
opendir	<dirent.h>	POSIX_DIRECTORY
openlog	<syslog.h>	SUS_SYSLOG.open
pathconf	<unistd.h>	POSIX_DIRECTORY.max_filename_length
pause	<unistd.h>	EPX_CURRENT_PROCESS.pause
perror	<stdio.h>	
pipe	<unistd.h>	EPX_PIPE.make
printf	<stdio.h>	
putc	<stdio.h>	
putchar	<stdio.h>	
puts	<stdio.h>	
raise	<signal.h>	STDC_SIGNAL.raise
rand	<stdlib.h>	STDC_CURRENT_PROCESS.random
read	<unistd.h>	EPX_FILE_DESCRIPTOR.read
readdir	<dirent.h>	POSIX_DIRECTORY
realloc	<stdlib.h>	STDC_BUFFER.resize
remove	<stdio.h>	POSIX_FILE_SYSTEM.remove_file
rename	<unistd.h>	POSIX_FILE_SYSTEM.rename_to
rewind	<stdio.h>	STDC_FILE.rewind
rewinddir	<dirent.h>	POSIX_DIRECTORY
rmdir	<unistd.h>	EPX_FILE_SYSTEM.remove_directory
scanf	<stdio.h>	
select	<sys/select.h>	[EPX_SELECT]
sem_close	<semaphore.h>	
sem_destroy	<semaphore.h>	
sem_getvalue	<semaphore.h>	
sem_init	<semaphore.h>	POSIX_UNNAMED_SEMAPHORE.create_shared
sem_open	<semaphore.h>	
sem_post	<semaphore.h>	POSIX_SEMAPHORE.release
sem_trywait	<semaphore.h>	POSIX_SEMAPHORE.attempt_acquire
sem_unlink	<semaphore.h>	
sem_wait	<semaphore.h>	POSIX_SEMAPHORE.acquire
setbuf	<stdio.h>	STDC_FILE.set_buffer
setgid	<unistd.h>	POSIX_CURRENT_PROCESS.set_group_id
setlocale	<locale.h>	STDC_CURRENT_PROCESS.set_locale
setpgid	<unistd.h>	PAPI_UNISTD.posix_setsid
setsid	<unistd.h>	PAPI_UNISTD.posix_setsid
setuid	<unistd.h>	POSIX_CURRENT_PROCESS.set_user_id
setvbuf	<stdio.h>	STDC_FILE.set_no_buffering
shm_open	<sys/mman.h>	POSIX_SHARED_MEMORY.open_read_write
shm_unlink	<sys/mman.h>	POSIX_FILE_SYSTEM.unlink_shared_memory_object
sigaction	<signal.h>	POSIX_SIGNAL
sigaddset	<signal.h>	POSIX_SIGNAL_SET.add
sigdelset	<signal.h>	POSIX_SIGNAL_SET.prune
sigemptyset	<signal.h>	POSIX_SIGNAL_SET.make_empty
sigfillset	<signal.h>	POSIX_SIGNAL_SET.make_full
sigismember	<signal.h>	POSIX_SIGNAL_SET.has

signal	<signal.h>	STDC_SIGNAL.raise
sigpending	<signal.h>	POSIX_SIGNAL_SET.make_pending
sigprocmask	<signal.h>	POSIX_SIGNAL_SET.add_to_blocked_signals
sigqueue	<signal.h>	
sigsuspend	<signal.h>	POSIX_SIGNAL_SET.suspend
sigtimedwait	<signal.h>	
sigwait	<signal.h>	
sigwaitinfo	<signal.h>	
sleep	<unistd.h>	POSIX_CURRENT_PROCESS.sleep
sprintf	<stdio.h>	
srand	<stdlib.h>	STDC_CURRENT_PROCESS.set_random_seed
sscanf	<stdio.h>	
stat	<sys/stat.h>	POSIX_STATUS
strftime	<time.h>	STDC_TIME.format
sysconf	<unistd.h>	POSIX_SYSTEM
syslog	<syslog.h>	SUS_SYSLOG
system	<stdlib.h>	STDC_SHELL_COMMAND
tcdrain	<unistd.h>	
tcflow	<unistd.h>	
tcflush	<unistd.h>	POSIX_TERMIOS.flush_input
tcgetattr	<unistd.h>	POSIX_TERMIOS.make
tcgetpgrp	<unistd.h>	
tcsendbreak	<unistd.h>	
tcsetattr	<unistd.h>	POSIX_TERMIOS.apply_now
tcsetpgrp	<unistd.h>	
time	<time.h>	STDC_TIME.make_from_unix_time
timer_create	<signal.h>	
timer_create	<time.h>	
times	<times.h>	
tmpfile	<stdio.h>	STDC_TEMPORARY_FILE.make
tmpnam	<stdio.h>	STDC_FILE_SYSTEM.temporary_file_name
ttyname	<unistd.h>	POSIX_FILE_DESCRIPTOR.ttyname
tzset	<time.h>	
umask	<sys/stat.h>	
uname	<sys/utsname.h>	POSIX_SYSTEM
ungetc	<stdio.h>	STDC_FILE.ungetc
unlink	<unistd.h>	POSIX_FILE_SYSTEM.unlink
utime	<utime.h>	POSIX_FILE_SYSTEM.utime
vfprintf	<stdio.h>	
vprintf	<stdio.h>	
vsprintf	<stdio.h>	
wait	<sys/wait.h>	POSIX_CURRENT_PROCESS.wait
waitpid	<sys/wait.h>	POSIX_FORK_ROOT.wait_pid
write	<unistd.h>	EPX_FILE_DESCRIPTOR.write

This tabel does not contain the following category of functions:

1. Math functions.
2. String functions, including wide character/multibyte string. routines. The memory move/copy functions are included, some of them even supported.
3. No type conversion functions.
4. No functions from <ctype.h>.
5. No functions from <setjmp.h>.

6. No functions from `<stdarg.h>`.
7. No string formatting functions like `sscanf`. I suggest you use the Formatter library for that.
You can download this library at <http://www.pobox.com/~berend/eiffel/>.

Functions in above categories are either not applicable, already present in Eiffel or are better off in a different library.

To do

ABSTRACT_DIRECTORY

1. `ABSTRACT_DIRECTORY.forth_recursive` raises an exception when it encounters a symbolic link that does no longer point to a file. Because it tries to retrieve the statistics, and that call fails.

EPX_FILE_SYSTEM

1. Make `EPX_DIRECTORY`.

STDC_FILE

1. `read_integer`, `read_double`, `read_boolean` should perhaps be different for the binary or text files. Now they're satisfy the mico/e definition, so useful for text files only.

STDC_LOCALE_NUMERIC

1. Complete the list of properties

STDC_PATH

1. make some escape char functionality with '%' or so.

STDC_TIME

1. Add elapsed seconds

POSIX_DAEMON

1. Closing the first three file descriptors is not likened by SmartEiffel. So leaves them open. Have to fix this some how.

POSIX_EXEC_PROCESS

1. Turn off Eiffel exception handling after the final `execvp`, else you get back signals not captured by child process as your signals, or so it seems (or perhaps you're killing the Eiffel process, but not the subprocess it generated??)

Killing subprocesses works sometimes, but not always.

Remove exception handling just before `execvp`?

2. how about capture to `/dev/null`?
3. can we capture i/o for every forked process? If so, move this code to `POSIX_FORK_ROOT`.
4. Perhaps option to influence environment variables to pass to subprocess?

POSIX_FILE_DESCRIPTOR

1. possible to open exclusively and so?
2. complete support for nonblocking i/o.

POSIX_MEMORY_MAP

1. Cannot change protection.
2. No locking.

POSIX_SEMAPHORE

1. not valid for named semaphore I think.
2. have to add various close/unlink functions.

POSIX_SIGNAL

1. Add synchronous waiting for signals like `sigwait`.
2. (Re)enable sending Eiffel exception on signal? i.e. `set_exception_handler` or so.
3. Resend signal as Eiffel exception in signal handler.

POSIX_STATUS

1. return `STDC_TIME` instead of unix time
2. Not all stat member fields are currently available.

POSIX_MQUEUE

1. Solaris x86 says it supports it, so have to work on that.

Security

Add base security class that specifies programs intent. Default is to allow anything, but security can be tightened:

1. Call to `open` or `creat` (used?), use real user id, not effective user id.
2. Assume we're free from buffer attacks if preconditions are enabled.
3. `exec/system` call only allowed when effective user is not root, unless otherwise specified. Or `exec` only allowed for specific files.
4. Protect against writing specific files/directories. Perhaps substitute vulnerable filenames for other ones.
5. Emulate atomic calls. Or add atomic `access` and `open` call. Shouldn't be done by setting `su??`
6. When appending/writing to files, check if symbolic link.
7. `ABSTRACT_FILE_SYSTEM.force_remove_directory` is potentially unsafe because it follows links so it can be used to destroy things not under that directory.
8. `remove tmpnam` function.
9. Make sure the once functions in `STDC_BASE` are called from within the security initialization, so they're allocated and do not generate an out-of-memory exception themselves.

Idea from 'Remediation of Application Specific Security Vulnerabilities at Runtime' article in IEEE Computer sep/oct 2000.

Windows code

1. `chmod` also available on Windows.
2. Add permissions to status: read/write.
3. `set_binary_mode` should do something for the posix factory, i.e., when compiling with cygwin. Perhaps separate `CYGWIN_API` or so in POSIX dir with the window specific stuff. Currently cygwin uses text mode for file descriptors, the windows variant uses binary.
4. `utime` can be supported by using `SetFileTime`.

Other

1. remove ugly `const_` prefix from constants. Uppercase should be good enough. Almost done, only `const_EOF` remains, not easy to replace perhaps.
2. Compare `POSIX_SIGNAL` with ISE `UNIX_SIGNAL`: They have an `is_caught` function, useful? Means this signal generates an exception.

Known bugs

- The error code is perhaps not always set for every `STDC_BASE.raise_posix_error`.
- does `STRING_HELPER` leak memory in `to_external`? How is memory used for these conversions being freed? Is memory used there?
- If a child process is signalled (terminated), the function `POSIX_FORK_ROOT.is_terminated_normally` sometimes returns True.

Bibliography

- 1 (1996). *System Application Program Interface (API) [C Language]*, volume Part I of *Information technology – Portable Operating System Interface (POSIX)*. ANSI/IEEE, 1996 edition.
- 2 (1991). *The Standard C library*. Prentice Hall.
- 3 (1994). *POSIX programmer's guide*. O'Reilly & Associates.
- 4 Stevens, W. R. (1998). *Unix network programming*. Prentice Hall.
- 5 Meyer, B. (1997). *Object-Oriented Software Construction*. Addison Wesley, 2nd edition.
- 6 Hatton, L. (2001). Exploring the role of diagnosis in software failure. *IEEE Software*.
- 7 Whittaker, J. A. (2001). Software's invisible users. *IEEE Software*.

Index

) 56
/src/library.xace 7, 8, 8
[5, 25, 66, 93
90
_exit 91

a

abort 90
abort
STDC_CURRENT_PROCESS 90
ABSTRACT_FILE_DESCRIPTOR 7, 17,
23, 33, 36
accept 90
accept
EPX_TCP_SERVER_SOCKET 90
access 90, 98
acquire
POSIX_SEMAPHORE 93
add
POSIX_SIGNAL_SET 93
add_data
EPX_CGI 74
add_raw
EPX_CGI 74
add_to_blocked_signals
POSIX_SIGNAL_SET 94
aio.h 90, 92
aio_cancel 90
aio_error 90
aio_fsync 90
aio_read 90
aio_return 90
aio_suspend 90
aio_write 90
alarm 90
allocate
STDC_BUFFER 92
allocate_and_clear
STDC_BUFFER 19, 90
ANY 5
apply
POSIX_SIGNAL 48

apply_drain
POSIX_TERMIOS 94
apply_flush
POSIX_TERMIOS 94
apply_now
POSIX_TERMIOS 94
apply_owner_and_group
POSIX_PERMISSIONS_PATH 90
arpa/inet.h 92
asctime 90
assert_key_value_pairs_created
EPX_CGI 77
atexit 90
attempt_acquire
POSIX_SEMAPHORE 93
attempt_lock
POSIX_FILE_DESCRIPTOR 91
attempt_open_read
POSIX_TEXT_FILE 80

b

b_a
EPX_CGI 74
b_form_get
EPX_CGI 75
b_form_post
EPX_CGI 75
b_input
EPX_CGI 75
b_p
EPX_CGI 74
backslash 27
BeOS 7
non-blocking i/o 7
big endian 20
binary file 27
binary mode 33
binary stdin 33
binary stdout 33
bind 90
browse_directory
POSIX_FILE_SYSTEM 42, 81

c`c_stdio.c` 88`c_stdio.h` 88`calloc` 90`cancel`

POSIX_ASYNC_IO_REQUEST 90

`CAPI_STDIO` 10, 88`C compiler`

Borland 2, 5

lcc 2

Microsoft 2

Microsoft Visual C

+ 5

`cecil.se` 6`cfgetispeed` 90`cfgetospeed` 90`cfsetispeed` 90`cfsetospeed` 90`cgi` 72

enumerating all values 78

file upload 74

redirect 78

`change_directory`

POSIX_FILE_SYSTEM 90

`change_mode`

POSIX_FILE_SYSTEM 90

`chdir` 90`chmod` 90`chown` 90`clear_error`

STDC_FILE 90

`clear_first`

STDC_ERRNO 83

`clearerr` 90`clock` 90`clock`

STDC_CURRENT_PROCESS 90

`clock_getcpuclockid` 90`clock_getres` 90`clock_gettime` 90`clock_nanosleep` 90`clock_settime` 90`close` 90`close`

EPX_FILE_DESCRIPTOR 90

EPX_SMTP_CLIENT 66

STDC_FILE 86, 91

SUS_SYSLOG 90

`closedir` 90`closelog` 90`compiler.se` 16`configure` 1, 7`confstr` 90`connect` 90`content_text_html`

EPX_CGI 77

`copy_from`

STDC_BUFFER 92

`creat` 90, 98`create_fifo`

POSIX_FILE_SYSTEM 16, 92

`create_read_write`

EPX_FILE_DESCRIPTOR 90

`create_shared`

POSIX_UNNAMED_SEMAPHORE 93

`create_unshared`

POSIX_UNNAMED_SEMAPHORE 93

`create_write`

POSIX_SHARED_MEMORY 93

`ctermid` 91`ctime` 91`Ctrl`

C 46, 48

`ctype.h` 94`current_directory`

POSIX_FILE_SYSTEM 92

`cuserid` 91`Cygwin` 8`cygwin` 10`CYGWIN` 33`CYGWIN_API` 98**d**`daylight` 91`deallocate`

STDC_BUFFER 91

`default_format`

POSIX_TIME 53

STDC_TIME 90

`detach`

POSIX_DAEMON 67

`difftime` 91`directory`

browse 42

change 40

create 40

- remove 40
- test_suite 18
- dirent.h 90, 93
- dispose
 - MEMORY 86
- doctype
 - EPX_XML_WRITER 73
- doctype_transitional
 - EPX_XML_WRITER 73
- dup 91
- dup2 91
- e**
- EEXIST 81
- effective_group_id
 - POSIX_CURRENT_PROCESS 92
- effective_user_id
 - POSIX_CURRENT_PROCESS 92
- ehlo
 - EPX_SMTP_CLIENT 65
- eiffel.h 87
- Eiffel Forum Freeware License v
- elj-win32 2
- endgrent 91
- endhostent 91
- endnetent 91
- endprotoent 91
- endpwent 91
- endservent 91
- ENOSYS 16
- environment variable 27
 - CFLAGS 8
 - CYGWIN 33
 - EPOSIX 1
- Environment variable
 - expansion 27
- environment variable
 - GOBO_CC 1, 2, 3
 - GOBO_EIFFEL 3
 - set 55
- eof
 - POSIX_TEXT_FILE 26
 - STDC_FILE 91
- EPX_CGI vi, 72
- EPX_CURRENT_PROCESS 17, 33, 47
- EPX_DIRECTORY 96
- EPX_EXEC_PROCESS 17
- EPX_FILE_DESCRIPTOR 16, 17
- EPX_FILE_SYSTEM 16, 17
- EPX_HTTP_10_CLIENT 62
- EPX_HTTP_SERVER 63
- EPX_LOG_HANDLER 69
- EPX_MIME_EMAIL 66
- EPX_PIPE 17
- EPX_SMTP_CLIENT 65
- EPX_SMTP_MAIL 65
- EPX_SOCKET 7
- EPX_SYSTEM 55
- EPX_TCP_CLIENT_SOCKET 7
- EPX_URI 63
- EPX_XHTML_WRITER 72
- epxc 11
- epxs 11
- errno 10
- errno
 - POSIX_FILE_DESCRIPTOR 82
- errno.first_value
 - POSIX_FILE_DESCRIPTOR 82
- errno.value
 - POSIX_FILE_DESCRIPTOR 82
- error
 - STDC_FILE 91
 - SUS_SYSLOG 94
- error handling 80
- EX_ERROR1 83
- execl 91
- execle 91
- execvp 91
- execute
 - EPX_CGI 72, 73, 74
 - EPX_EXEC_PROCESS 91
 - POSIX_DAEMON 67
 - POSIX_FORK_ROOT 50
 - POSIX_SHELL_COMMAND 45
- execv 91
- execve 91
- execvp 91
- exit 91
- exit
 - STDC_CURRENT_PROCESS 91
- exit_switch
 - STDC_EXIT_SWITCH_ACCESSOR 90
- EXP_FTP_CLIENT 62
- expand_path
 - POSIX_FILE_SYSTEM 27

f

`fchmod` 91
`fchown` 91
`fclose` 91
`fcntl` 91
`fcntl.h` 90, 93
fd_stdin
 EPX_CURRENT_PROCESS 33
fd_stdout
 EPX_CURRENT_PROCESS 33
`fdatasync` 7, 7, 7, 91
`fdopen` 91
`feof` 91
`ferror` 91
`fflush` 91
`fgetc` 91, 92
`fgetpos` 91
`fgets` 91, 92
file
 EPX_KEYVALUE 77
file
 read entire 25
 filename manipulation 38
 fileno 91
 file pointer 28
fill_with
 STDC_BUFFER 92
first_value
 POSIX_FILE_DESCRIPTOR 82
 STDC_ERRNO 83
`flockfile` 91
`flush` 34
flush
 STDC_FILE 91
flush_input
 POSIX_TERMIOS 94
`fopen` 87, 91
force_remove_directory
 ABSTRACT_FILE_SYSTEM 98
`fork` 91
fork
 POSIX_CURRENT_PROCESS 50, 91
format
 POSIX_TIME 53
 STDC_TIME 94
forth_recursive
 ABSTRACT_DIRECTORY 96
`forum.txt` v

`fpathconf` 91
`fprintf` 91
`fputc` 91, 93
`fputs` 91, 93
`fread` 91
`free` 91
`freopen` 91
`fseek` 91
`fsetpos` 91
`fstat` 91
`fsync` 7, 7, 7, 91
`ftell` 91
`ftruncate` 92
`ftrylockfile` 92
`funlockfile` 92
`fwrite` 92

g

`geant` 1
get_character
 STDC_FILE 91
get_lock
 POSIX_FILE_DESCRIPTOR 16, 31, 91
get_position
 POSIX_FILE 28
 STDC_FILE 91
get_string
 STDC_FILE 91
`getc` 92
`getchar` 92
`getcwd` 92
`getegid` 92
`getenv` 92
`geteuid` 92
`getgid` 92
`getgrgid` 92
`getgrnam` 92
`getgroups` 92
`getlogin` 91, 92
`getpgrp` 92
`getpid` 14, 92
`getppid` 92
`getpwnam` 92
`getpwuid` 92
`gets` 92
`gettimeofday` 92
`getuid` 92
`gexace` 4

gmtime 92
Gobo 23, 36
grp.h 91, 92

h

has
 POSIX_SIGNAL_SET 93
htonl 92
htons 92
HTTP 11

i

inet_ntoa 92
input_speed
 POSIX_TERMIOS 90
input_text
 EPX_CGI 75
install
 STDC_EXIT_SWITCH 90
IRC_CLIENT 64
is_accessible
 ABSTRACT_FILE_SYSTEM 90
is_attached_to_terminal
 EPX_FILE_DESCRIPTOR 92
is_blocking_io
 ABSTRACT_FILE_DESCRIPTOR 36
is_in_group
 POSIX_CURRENT_PROCESS 92
is_modifiable
 POSIX_FILE_SYSTEM 40
is_pending
 POSIX_ASYNC_IO_REQUEST 90
is_readable
 POSIX_FILE_SYSTEM 41
is_terminated_normally
 POSIX_FORK_ROOT 98
isatty 92
ISE Eiffel 2

k

KI_CHARACTER_INPUT_STREAM 23, 36
KI_CHARACTER_OUTPUT_STREAM 23
kill 92
kill
 POSIX_PROCESS 92

l

last_string
 POSIX_TEXT_FILE 26
libposix_ise_msc.lib 3
libposix_se.a 2, 16
libposix_ve_msc.lib 6
library.xace 4
license v
link 92
link
 POSIX_FILE_SYSTEM 92
lio_listio 92
listen_by_address
 EPX_TCP_SERVER_SOCKET 90
little endian 20
local_date_string
 POSIX_TIME 53
local_time_string
 POSIX_TIME 53
locale.h 92, 93
localeconv 92
localtime 92
lock 30
log_event
 ULM_LOGGING 70
log_message
 ULM_LOGGING 70
login_name
 POSIX_CURRENT_PROCESS 92
lseek 92

m

mail
 EPX_SMTP_CLIENT 65
make
 EPX_LOG_HANDLER 71
 EPX_PIPE 93
 POSIX_TERMIOS 94
 STDC_TEMPORARY_FILE 94
make.exe 2
make_as_duplicate
 EPX_FILE_DESCRIPTOR 91
 POSIX_FILE_DESCRIPTOR 34
make_directory
 EXP_FTP_CLIENT 62
 POSIX_FILE_SYSTEM 92
make_empty
 POSIX_SIGNAL_SET 93

make_expand
 STDC_PATH 40
make_from_file
 POSIX_FILE_DESCRIPTOR 91
make_from_file_descriptor
 POSIX_FILE 91
make_from_gid
 POSIX_GROUP 92
make_from_name
 POSIX_GROUP 92
 POSIX_USER 92
make_from_now
 POSIX_TIME 53
make_from_uid
 POSIX_USER 92
make_from_unix_time
 STDC_TIME 94
make_full
 POSIX_SIGNAL_SET 93
make_pending
 POSIX_SIGNAL_SET 94
malloc 92
max_filename_length
 POSIX_DIRECTORY 93
max_open_files 55
memchr 92
memcmp 92
memcpy 92
memmove 92
memory_copy
 STDC_BUFFER 92
memory_move
 STDC_BUFFER 92
memset 92
MIME 11
minicom 34
mkdir 92
mkfifo 7, 7, 16, 92
mktime 92
mlock 92
mlockall 92
mmap 92
modem 34
mprotect 92
mq-receive 92
mq_close 92
mq_getattr 92
mq_notify 92
mq_open 92
mq_send 92
mq_setattr 92
mq_unlink 92
mqueue.h 92
msync 92
multi-threaded programming 2
munlock 93
munlockall 92
munmap 93

n
nanosleep 93
netdb.h 91
netinet/in.h 92, 93
non-blocking i/o 23, 36
notice
 SUS_SYSLOG 94
ntohl 93
ntohs 93

o
open 93, 98
open
 EPX_FILE_DESCRIPTOR 93
 POSIX_FILE 10
 SUS_SYSLOG 93
open_by_address
 EPX_TCP_CLIENT_SOCKET 90
open_by_name_and_port
 EPX_TCP_CLIENT_SOCKET 90
open_read
 EPX_FILE_DESCRIPTOR 93
 POSIX_FILE 10
 POSIX_SHARED_MEMORY 93
 POSIX_TEXT_FILE 80
open_read_write
 EPX_FILE_DESCRIPTOR 93
 POSIX_SHARED_MEMORY 93
open_write
 EPX_FILE_DESCRIPTOR 93
opendir 93
openlog 93
Open Source v
out
 EPX_IP4_ADDRESS 92
output_speed
 POSIX_TERMIOS 90

P

- `p_stdio.c` 88
 - `p_stdio.h` 88
 - `PAPI_UNISTD` 10
 - `parent_pid`
 - `POSIX_CURRENT_PROCESS` 92
 - `parse`
 - `STDC_PATH` 40
 - `pathconf` 93
 - `path name` 27
 - `pause` 93
 - `pause`
 - `EPX_CURRENT_PROCESS` 93
 - `peek_int16`
 - `STDC_BUFFER` 20
 - `peek_int16_big_endian`
 - `STDC_BUFFER` 20
 - `peek_int16_little_endian`
 - `STDC_BUFFER` 20
 - `peek_int32`
 - `STDC_BUFFER` 20
 - `peek_uint16`
 - `STDC_BUFFER` 20
 - `permissions`
 - `POSIX_FILE_SYSTEM` 41
 - `perror` 93
 - `pid`
 - `POSIX_CURRENT_PROCESS` 14, 92
 - `pipe` 93
 - `poke_int32_big_endian`
 - `STDC_BUFFER` 20
 - `poll` v
 - `POSIX_ASYNC_IO_REQUEST` 37
 - `POSIX_BASE` 10
 - `POSIX_BINARY_FILE` 23
 - `POSIX_BUFFER` 19, 19, 20
 - `POSIX_CONSTANTS` 11
 - `POSIX_CURRENT_PROCESS` 50
 - `POSIX_DAEMON` 67, 67
 - `POSIX_DIRECTORY` 42, 43, 90, 93
 - `POSIX_EXEC_PROCESS` vi, 45
 - `POSIX_FILE` 23, 23
 - `POSIX_FILE_DESCRIPTOR` 17, 29, 86, 91
 - `POSIX_FILE_SYSTEM` 40
 - `POSIX_FORK_ROOT` 14, 50
 - `posix_htonl`
 - `SAPI_IN` 92
 - `posix_htons`
 - `SAPI_IN` 92
 - `posix_memcmp`
 - `CAPI_STRING` 92
 - `POSIX_MEMORY_MAP` 21, 92, 93
 - `posix_ntohl`
 - `SAPI_IN` 93
 - `posix_ntohs`
 - `SAPI_IN` 93
 - `POSIX_PERMISSIONS` 41, 42
 - `posix_setsid`
 - `PAPI_UNISTD` 93
 - `POSIX_SHARED_MEMORY` 19
 - `POSIX_SHELL_COMMAND` 45
 - `POSIX_SIGNAL` 93
 - `POSIX_SIGNAL_HANDLER` 48, 49
 - `POSIX_STAT` 42
 - `POSIX_STATUS` 42, 91, 94
 - `POSIX_SYSTEM` 55, 94
 - `POSIX_TEXT_FILE` 23, 30
 - `POSIX_TIMED_COMMAND` 90
 - `printf` 93
 - `process_group_id`
 - `POSIX_CURRENT_PROCESS` 92
 - `prune`
 - `POSIX_SIGNAL_SET` 93
 - `put_string`
 - `POSIX_FILE_DESCRIPTOR` 82, 83
 - `STDC_FILE` 91
 - `putc` 93
 - `putc`
 - `STDC_FILE` 91
 - `putchar` 93
 - `puts` 93
 - `puts`
 - `EPX_CGI` 74
 - `pwd.h` 91, 92
- q**
- `QNX` 7
 - `quit`
 - `EPX_SMTP_CLIENT` 66
- r**
- `raise` 93
 - `raise`
 - `STDC_SIGNAL` 93, 94

raise_posix_error
 STDC_BASE 98

rand 93

random
 STDC_CURRENT_PROCESS 93

raw_value
 EPX_CGI 75

read 7, 93

read
 ABSTRACT_FILE_DESCRIPTOR 7, 36
 EPX_FILE_DESCRIPTOR 93
 POSIX_ASYNC_IO_REQUEST 37, 90
 POSIX_FILE 26
 STDC_FILE 91

read_buffer
 POSIX_FILE 26

read_character
 STDC_FILE 91

read_line
 [25
 ABSTRACT_FILE_DESCRIPTOR 33

read_string
 [25
 ABSTRACT_FILE_DESCRIPTOR 23
 POSIX_TEXT_FILE 26
 STDC_FILE 91

readdir 93

real_group_id
 POSIX_CURRENT_PROCESS 92

real_time_clock
 SUS_SYSTEM 90

real_time_clock_resolution
 SUS_SYSTEM 90

real_user_id
 POSIX_CURRENT_PROCESS 92

realloc 93

recv 7

redirect standard error 34

reestablish
 STDC_SIGNAL_HANDLER 49

refresh
 POSIX_PERMISSIONS 42

release
 POSIX_SEMAPHORE 93

remove 93

remove_directory
 EPX_FILE_SYSTEM 93
 EXP_FTP_CLIENT 62

remove_file
 EXP_FTP_CLIENT 62
 GENERAL 5
 POSIX_FILE_SYSTEM 5, 81, 93

remove_from_blocked_signals
 POSIX_SIGNAL_SET 94

rename 93

rename_to
 EXP_FTP_CLIENT 62
 POSIX_FILE_SYSTEM 93

reopen
 STDC_FILE 91

resize
 STDC_BUFFER 93

restore_group_id
 POSIX_CURRENT_PROCESS 93

restore_user_id
 POSIX_CURRENT_PROCESS 93

return_status
 POSIX_ASYNC_IO_REQUEST 90

rewind 93

rewind
 STDC_FILE 93

rewinddir 93

rmdir 93

s

save_uploaded_files
 EX_CGI3 77

scanf 93

security.cpu.check_process_time
 STDC_FILE 86

security.cpu.set_max_process_time
 STDC_FILE 86

security.error_handling.disable_exceptions
 STDC_SECURITY_ACCESSOR 82

security.error_handling.enable_exceptions
 STDC_SECURITY_ACCESSOR 82

security.files.set_max_open_files
 STRING 86

security.memory.set_max_allocation
 STDC_SECURITY_ACCESSOR 85

security.memory.set_max_single_allocation
 STRING 86

seek 28

seek
 EPX_FILE_DESCRIPTOR 92
 POSIX_FILE 28

- STDC_FILE 91
- seek_from_current*
 - EPX_FILE_DESCRIPTOR 92
 - STDC_FILE 91
- seek_from_end*
 - EPX_FILE_DESCRIPTOR 92
 - STDC_FILE 91
- select* v, 93
- sem_close* 93
- sem_destroy* 93
- sem_getvalue* 93
- sem_init* 93
- sem_open* 93
- sem_post* 93
- sem_trywait* 93
- sem_unlink* 93
- sem_wait* 93
- semaphore.h* 93
- sendmsg* 7
- set_allow_anyone_read*
 - POSIX_PERMISSIONS 42
- set_allow_group_write*
 - POSIX_PERMISSIONS 42
- set_blocked_signals*
 - POSIX_SIGNAL_SET 94
- set_blocking_io*
 - ABSTRACT_FILE_DESCRIPTOR 36
- set_buffer*
 - POSIX_ASYNC_IO_REQUEST 37
 - STDC_FILE 93
- set_count*
 - POSIX_ASYNC_IO_REQUEST 37
- set_date*
 - STDC_TIME 92
- set_date_time*
 - STDC_TIME 92
- set_full_buffering*
 - STDC_FILE 93
- set_group_id*
 - POSIX_CURRENT_PROCESS 93
- set_handler*
 - POSIX_SIGNAL 48, 49
- set_input_speed*
 - POSIX_TERMIOS 90
- set_line_buffering*
 - STDC_FILE 93
- set_locale*
 - STDC_CURRENT_PROCESS 93
- set_lock*
 - POSIX_FILE_DESCRIPTOR 91
- set_native_locale*
 - STDC_CURRENT_PROCESS 93
- set_native_time*
 - STDC_CURRENT_PROCESS 93
- set_no_buffering*
 - STDC_FILE 93
- set_offset*
 - POSIX_ASYNC_IO_REQUEST 37
- set_output_speed*
 - POSIX_TERMIOS 90
- set_position*
 - POSIX_FILE 28
 - STDC_FILE 91
- set_random_seed*
 - STDC_CURRENT_PROCESS 94
- set_time*
 - STDC_TIME 92
- set_user_id*
 - POSIX_CURRENT_PROCESS 93
- setbuf* 93
- setgid* 93
- setjmp.h* 94
- setlocale* 93
- setpgid* 93
- setsid* 93
- setuid* 93
- setvbuf* 93
- shm_open* 93
- shm_unlink* 93
- sigaction* 93
- sigaddset* 93
- SIGCHLD** 49
- sigdelset* 93
- sigemptyset* 93
- sigfillset* 93
- sigismember* 93
- signal* 94
- signal.h* 92, 93, 94
- signal handler* 48
- signalled*
 - POSIX_SIGNAL_HANDLER 48, 49
- sigpending* 94
- sigprocmask* 94
- sigqueue* 94
- sigsuspend* 94
- sigtimedwait* 94

- sigwait 94, 97
- sigwaitinfo 94
- slash 27
- sleep 94
- sleep
 - EPX_CURRENT_PROCESS 47
 - POSIX_CURRENT_PROCESS 94
- SmallEiffel vi
- SmartEiffel 2
- sprintf 94
- srand 94
- src/library.xace 1, 6
- sscanf 94, 95
- stat 42, 94
- status
 - EPX_FILE_DESCRIPTOR 91
 - POSIX_FILE_DESCRIPTOR 42
- STC_TEMPORARY_FILE 15
- stdarg.h 95
- STDC_BASE 10
- STDC_BINARY_FILE 15, 33
- STDC_BUFFER 15, 19, 19, 85
- STDC_CONSTANTS 11, 15
- STDC_CURRENT_PROCESS 15
- STDC_ENV_VAR 15, 54
- STDC_ERRNO
 - POSIX_FILE_DESCRIPTOR 82
- STDC_FILE 23, 86, 91
- STDC_FILE_SYSTEM 15
- STDC_LOCALE_NUMERIC 92
- STDC_PATH 38
- STDC_SECURITY_ACCESSOR 85
- STDC_SHELL_COMMAND 15, 94
- stdc_signal_switch_switcher 6
- STDC_SYSTEM 15
- STDC_TEXT_FILE 15, 33
- STDC_TIME 15, 91
- stderr 34
- stdin
 - binary 33
- stdin
 - EPX_CURRENT_PROCESS 33
- stdio.h 88, 88, 90, 91, 92, 93, 94
- stdioh 92
- stdlib.h 90, 91, 92, 93, 94
- stdout 34
 - binary 33
- stdout
 - EPX_CURRENT_PROCESS 33
- store
 - EXP_FTP_CLIENT 62
- stream buffer 34
- strftime 94
- STRING 85
- string.h 92
- support
 - commercial v
- supports_nonblocking_io
 - ABSTRACT_FILE_DESCRIPTOR 7, 36
- SUS_BASE 10
- SUS_ENV_VAR
 - STDC_ENV_VAR 55
- SUS_SYSLOG 69, 94
- SUS_SYSLOG_ACCESSOR 69
- SUS_TIME_VALUE 92
- suspend
 - POSIX_SIGNAL_SET 94
- synchronize
 - POSIX_ASYNC_IO_REQUEST 37, 90
 - POSIX_FILE_DESCRIPTOR 91
- synchronize_data
 - POSIX_FILE_DESCRIPTOR 91
- sys/mman.h 92, 93
- sys/select.h 93
- sys/socket.h 90
- sys/stat.h 90, 91, 92, 94
- sys/time.h 92
- sys/utsname.h 94
- sys/wait.h 94
- sysconf 94
- syslog 94
- syslog.h 90, 93, 94
- system 94
- system.se 16
- system.xace 4, 6
- t
 - tcdrain 94
 - tcflow 94
 - tcflush 94
 - tcgetattr 94
 - tcgetpgrp 94
 - tcsendbreak 94
 - tcsetattr 94
 - tcsetpgrp 94

- tell*
 - POSIX_FILE 28
 - STDC_FILE 91
- temporary_file_name*
 - STDC_FILE_SYSTEM 94
- terminal* 31
 - password 31
- termios.h* 90
- TEST_IRC_CLIENT 64
- text mode 33
- time 94
- time.h 90, 91, 92, 93, 94
- timer_create 94
- times 94
- times.h 94
- tmpfile 94
- tmpnam 94
- to_local*
 - POSIX_TIME 53
 - STDC_TIME 92
- to_utc*
 - POSIX_TIME 53
 - STDC_TIME 92
- touch*
 - POSIX_FILE_SYSTEM 94
- ttyname* 94
- ttynam*
 - POSIX_FILE_DESCRIPTOR 94
- tzset* 94
- u**
 - ULM_LOG_HANDLER 69
 - ULM_LOG_LEVELS 70
 - ULM_LOGGING 69
 - umask 94
 - uname 94
 - unfinished_xml*
 - EPX_XML_WRITER 73
 - ungetc 94
 - ungetc*
 - STDC_FILE 94
 - unistd.h 90, 91, 92, 93, 94
 - unlink 10, 94
 - POSIX_FILE_SYSTEM 94
 - unlink_shared_memory_object*
 - POSIX_FILE_SYSTEM 93
- URI 11, 63
- utime 94
- utime*
 - POSIX_FILE_SYSTEM 94
- utime.h 94
- v**
 - value*
 - EPX_CGI 75, 77
 - STDC_ENV_VAR 92
 - VE_BIN 6
 - vfprintf 94
 - Visual Eiffel vi
 - VisualEiffel 2, 6
 - vprintf 94
 - vsprint 94
- w**
 - wait 94
 - wait*
 - POSIX_CURRENT_PROCESS 14, 94
 - wait_for*
 - POSIX_ASYNC_IO_REQUEST 37, 90
 - POSIX_CHILD 14
 - POSIX_EXEC_PROCESS 46
 - wait_pid*
 - POSIX_FORK_ROOT 94
 - waited_child_pid*
 - POSIX_CURRENT_PROCESS 14
 - waitpid 94
 - Windows 2000 8
 - WINDOWS_PAGING_FILE_SHARED_MEMORY 21
 - WINDOWS_SYSTEM 55
 - write 7, 94
 - write*
 - ABSTRACT_FILE_DESCRIPTOR 7, 36
 - EPX_FILE_DESCRIPTOR 94
 - POSIX_ASYNC_IO_REQUEST 37, 90
 - STDC_FILE 92
- x**
 - XM_UNICODE_CHARACTER_CLASSES 6

