

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	3
2	Using e-POSIX	4
2.1	Using <code>library.xace</code>	4
2.2	Vendor specific notes	5
2.2.1	Gobo Eiffel	5
2.2.2	ISE Eiffel	5
2.2.3	SmartEiffel	5
2.3	Platform specific notes	6
2.3.1	Linux	6
2.3.2	FreeBSD	6
2.3.3	Cygwin	6
2.3.4	Solaris	6
2.3.5	Win32	6
3	Design notes	7
3.1	Why an entire reimplementation?	7
3.2	Goals and guidelines	7
3.3	Class structure	8
3.4	Clients of this library	10
3.5	Forking	10
3.6	Books	12
4	Layers	13
4.1	Layers architecture	13
4.2	Standard C	13
4.3	Windows	13
4.3.1	Writing portable programs	13
4.3.2	Compiling POSIX programs in Windows	14
4.3.3	Native Windows	14
4.4	Introduction to the next chapters	16
5	Working with memory	17
5.1	Introduction	17
5.2	Allocating memory	17
5.3	Allocating memory	18
5.4	Using shared memory	18
5.5	Memory maps	19

6	Working with files	21
6.1	Introduction	21
6.2	Standard C notes	21
6.3	Compatibility with Gobo	21
6.4	Working with streams	21
6.5	Working with streams using Standard C only	26
6.6	Working with file descriptors	27
6.7	Windows systems: binary mode versus text mode	30
7	Working with files: advanced topics	32
7.1	Redirecting stderr to stdout	32
7.2	Talking to your modem	32
7.3	Non-blocking I/O	34
7.4	Asynchronous I/O	34
8	Working with the file system	36
8.1	Portability	36
8.2	Standard C	36
8.3	POSIX	38
9	Working with processes	42
9.1	Introduction	42
9.2	Executing a child command	42
9.3	Reading stdout of a child process	42
9.4	Catching a signal with Standard C	45
9.5	Catching a signal with POSIX	46
9.6	General wait for child handler	47
9.7	Forking a child process	48
10	Querying the operating system	51
10.1	Current time	51
10.2	Accessing environment variables	52
10.3	Capabilities	53
11	Working with the network	54
11.1	MIME parsing	54
11.2	Sockets	54
11.3	Echo client	54
11.4	Echo client and server	56
12	Working with the network: advanced topics	59
12.1	Introduction	59
12.2	FTP client	59
12.3	HTTP client	60
12.4	HTTP server	61
12.5	IMAP4 client	65
12.6	IRC client	66
12.7	SMTP client	67
12.7.1	Sending plain text email	67
12.7.2	Sending HTML email	69
12.7.3	Sending both text and HTML email	70

12.7.4	Sending attachments	71
12.8	LDIF parser	73
13	Writing daemons	74
13.1	Introduction	74
13.2	Windows	74
13.3	Creating a daemon	74
13.4	Logging messages and errors	75
13.5	ULM based logging	76
14	Writing CGI programs	79
15	Error handling	86
15.1	Error handling with exceptions	86
15.2	Manual error handling	88
16	Security	91
16.1	Denial of service attacks	91
16.2	Authorization bypass attacks	92
17	Accessing C headers	93
17.1	Making C Headers available to Eiffel	93
17.2	Distinction between Standard C and POSIX headers	94
17.3	C translation details	95
A	Posix function to Eiffel class mapping list	96
	To do	102
	<i>ABSTRACT_DIRECTORY</i>	102
	<i>EPX_FILE_SYSTEM</i>	102
	<i>STDC_FILE</i>	102
	<i>STDC_LOCALE_NUMERIC</i>	102
	<i>STDC_PATH</i>	102
	<i>STDC_TIME</i>	102
	<i>POSIX_DAEMON</i>	102
	<i>POSIX_EXEC_PROCESS</i>	102
	<i>POSIX_FILE_DESCRIPTOR</i>	103
	<i>POSIX_MEMORY_MAP</i>	103
	<i>POSIX_SEMAPHORE</i>	103
	<i>POSIX_SIGNAL</i>	103
	<i>POSIX_STATUS</i>	103
	<i>POSIX_MQUEUE</i>	103
	Security	103
	Windows code	104
	Other	104
	Known bugs	104
	Bibliography	105

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 3.0. And my to do list hasn't shrunked, so stay tuned!

I actively support this library, so bug reports and wishes are gladly accepted. Planned extensions are 64 bit integer support in every place, so you won't be limited to files of 2GB in size. And of course, more and more support for the remaining functions in the Single Unix Specification not yet covered, such as `poll`. On the protocol side I like to have NNTP server support. And perhaps one day we'll have native SSL!

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

Licensing

This software is licensed under the MIT License. This license can be found in the `LICENSE` 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 particularly 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 reportexd 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 e-POSIX which will lead to even closer integration with Gobo in subsequent releases.
- **[sven]**: various comments and suggestions.
- Colin Paul Adams: contributed classes such as the resolvers and fixes.
- Till G. Bay: contributed multiplexing support for e-POSIX's socket class.

Colophon

The text of this manual was entered with GNU Emacs 21.4.2 on BLinux. It was typeset with pdfTEX using the ConTEXt macro package, see <http://www.pragma-ade.com>. BON diagrams were created with METAPOST.

Requirements and installation

In this chapter:

- 1.1 Requirements*
- 1.2 Compiling the C code*

1.1 Requirements

e-POSIX has three requirements:

1. e-POSIX requires Gobo release 3.8 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`.

If you develop multi-thread applications, enable thread support with:

```
./configure --with-compiler=ise --prefix=$EPOSIX --enable-threads
make clean
make
make install
```


You can install both the multi-threaded library and the single thread library at the same time.

In addition you have to make sure the `GOBO_MT` environment variable is defined when generating the `.ecf` or `.ace` file. This requires a change to Gobo's `eifel.eant` file.

In the target for `xace` add this line:

```
<define name="GOBO_MT" value="mt" if="${GOBO_MT}"/>
```

You can now pass the `GOBO_MT` variable like this:

```
geant -D "GOBO_MT=mt" compile_ise
```

In case your applications are always multi-threaded, it might be easier to copy `libeposixmt_ise.a` to `libeposix_ise.a` so you don't have to bother with this. The sole purpose of `GOBO_MT` is to pick `libeposixmt_ise.a` or `libeposix_ise.a`.

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.

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>	<code>lcc-win32</code> compiler.

Table 1.1 Possible values for the `GOBO_CC` environment variable

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

```
makelib -ise -msc -mt
```

You must pass the `-mt` flag for ISE Eiffel if you are using the Microsoft Visual C compiler. You also will have to copy the multi-threaded library to the single-threaded library:

```
cd lib
copy libmteposix_ise_msc.lib libposix_ise_msc.lib
```

This is only supported for the ISE Eiffel compiler. e-POSIX 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 `libposix`. On Unix the name of the Eiffel vendor is appended, so `libposix_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:

2.1 *Using library.xace*
2.2 *Vendor specific notes*
2.3 *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 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) 2002-2007, 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="${GOBO}/library/string/library.xace"/>
<mount location="${GOBO}/library/time/library.xace"/>
<mount location="${GOBO}/library/utility/library.xace"/>
<mount location="${GOBO}/library/kernel.xace"/>

</system>

```

2.2 Vendor specific notes

2.2.1 Gobo Eiffel

e-POSIX supports the Gobo Eiffel compiler of Gobo version 3.8 or higher. Multi-threading has not been tested with this compiler.

2.2.2 ISE Eiffel

e-POSIX supports ISE Eiffel 6.2 and higher. e-POSIX has been tested under the following conditions:

1. I used Microsoft Windows 2000, Service Pack 2.
2. I used the included mingw compiler.

Note that you need the multithreaded version of the C binding library if you use ISE Eiffel and the Microsoft Visual C compiler. Else you will get a linker message complaining about the unresolved external symbol `_errno`.

You can also use the supplied `eposix.ecf` which will make inclusion of eposix in your projects much easier.

2.2.3 SmartEiffel

e-POSIX was tested with SmartEiffel 1.2r7 on FreeBSD, Linux, 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.

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:

¹ I wrote that two years ago...

```
-- 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 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 might 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 Windows and Linux only. If you use any other platform, you will have to edit `src/library.xace`.

2.3.1 Linux

The latest version of e-POSIX was tested on Ubuntu 8.0.4. and glibc 2.4.

2.3.2 FreeBSD

The latest version of e-POSIX was tested with FreeBSD 6.2-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`.

2.3.3 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.

2.3.4 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.

2.3.5 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](#).



Figure 3.1 e-POSIX
directory structure

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.

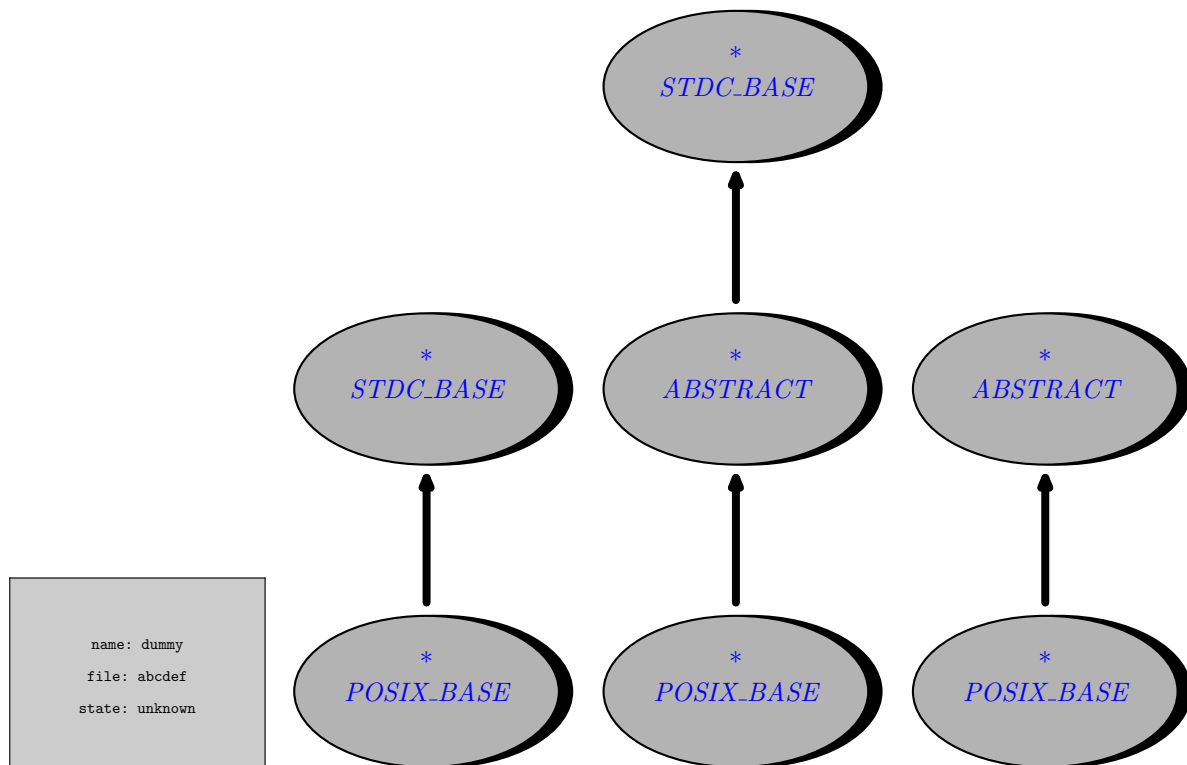


Figure 3.2 Inheritance structure

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.

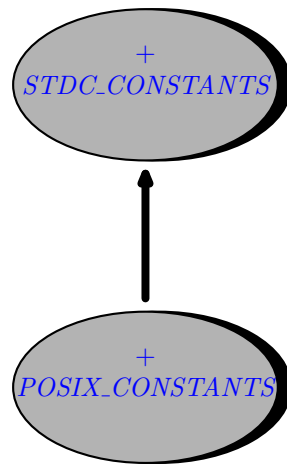


Figure 3.3 Standard C and POSIX constants

3.5 Forking

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

```

class PROCESS

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

```

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
end

```

```

execute is
  do
    -- forked code
  end
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 (XXXXXXXXXX, 0000), (XXXXXXXXXX, 0000) and (XXXXXXXXXX, 0000), see the biography section at [page 105](#).

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 these 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.

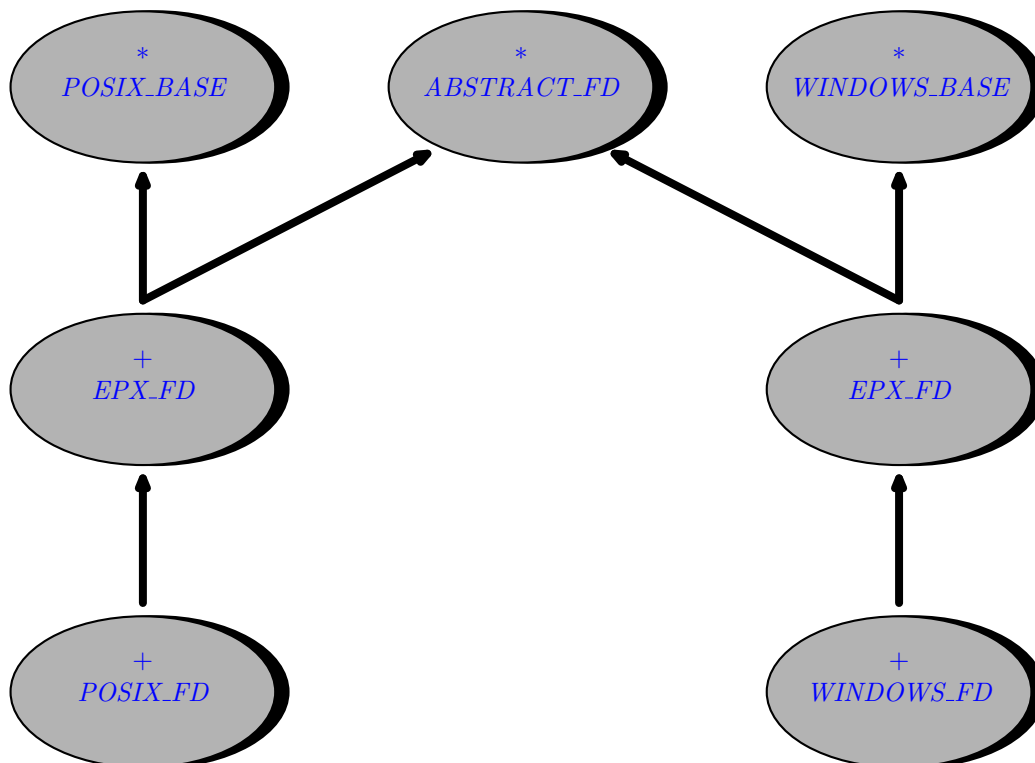


Figure 4.1 How EPX_XXXX classes are related to the POSIX and Windows classes

An example of using the `EPX _FILE _SYSTEM` class is shown below:

```

class EX_EPX1

inherit

    EPX_FILE_SYSTEM

create


    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 mem- ory

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

create

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

  create

    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

  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

  create
```

```

    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

```

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

create

    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](#).

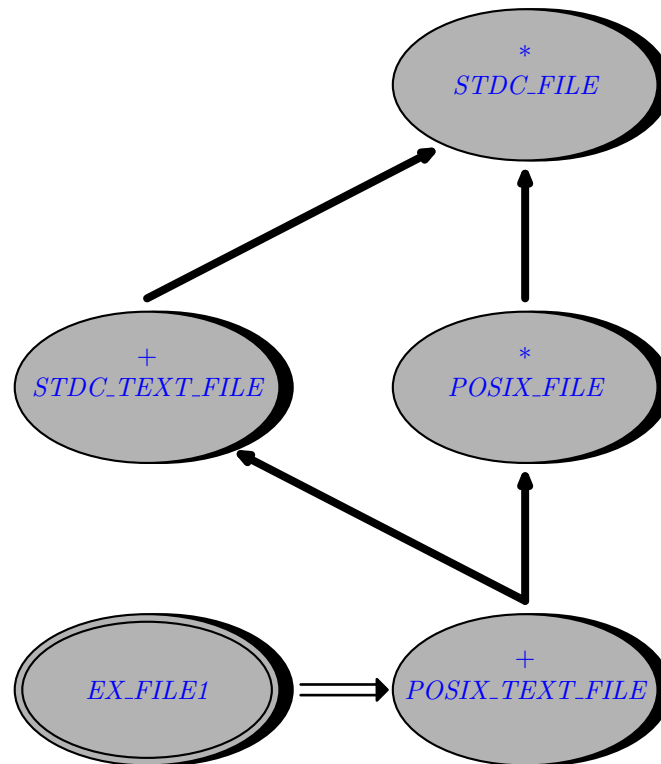


Figure 6.1 BON diagram of opening a text file.

```

class EX_FILE1

create

make

feature

make is
local
  file: POSIX_TEXT_FILE
do
  create file.open_read ("/etc/group")
  from
    file.read_line
  until
    file.end_of_input
  loop
    print (file.last_string)
    print ("%N")
    file.read_line
  end
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 e-POSIX 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

create

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.end_of_input
    loop
        file.read_buffer (buffer, 0, chunk_size)
    end
    file.close
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

  create

    make

  feature

    make is
    local
      file: POSIX_TEXT_FILE
    do
      create file.open_read ("/etc/group")
      from
      until
        file.end_of_input
      loop
        file.read_string (256)
        print (file.last_string)
      end
      file.close
    end

  end
```

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

```
class EX_WRONG2

  create

    make

  feature

    make is
    local
      file: POSIX_TEXT_FILE
    do
      create file.open_read ("/etc/group")
      from
      until
        file.end_of_input
      loop
```

```

    file.read_string (256)
  if not file.end_of_input 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

create

  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

create

```



```

    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

create

    make

feature

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

```

```

    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](#).

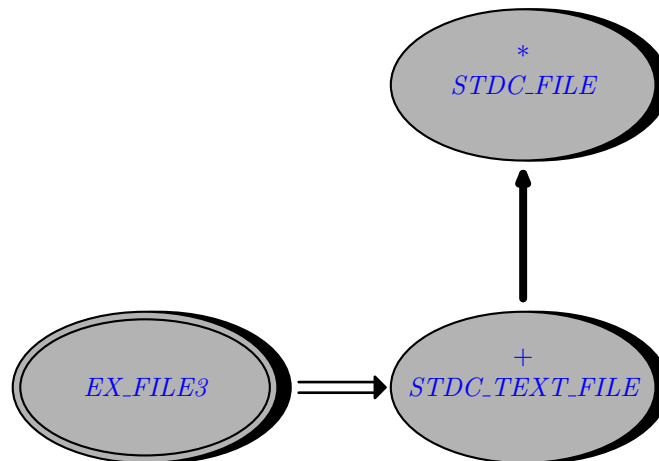


Figure 6.2 BON diagram of opening a Standard C text file.

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_FD1

create

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

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

create

    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.end_of_input
        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

create

    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.

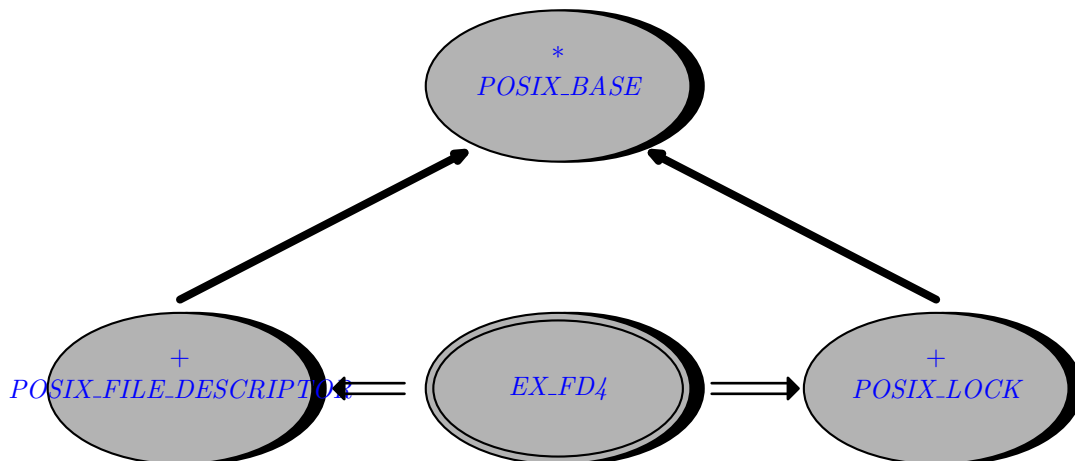


Figure 6.3 BON diagram of locking a portion of a file.

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

create

    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.

7

Working with files: advanced topics

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**

7.1 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_REDIRECT1
```

```
inherit
```

```
    POSIX_CURRENT_PROCESS
```

```
create
```

```
    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

create

    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 ("ATI0%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

```

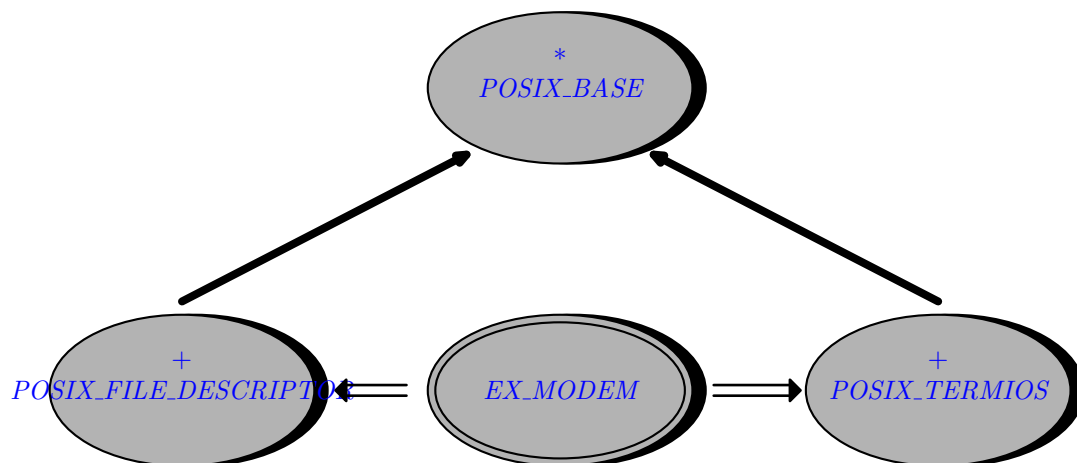


Figure 7.1 BON diagram of talking to a modem.

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

create

```

```
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

create

    make

feature

    make is
    do
        rename__to ("qctest.abc.tmp", "qctest.xyz.tmp")
        remove__file ("qctest.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

create

    make
```

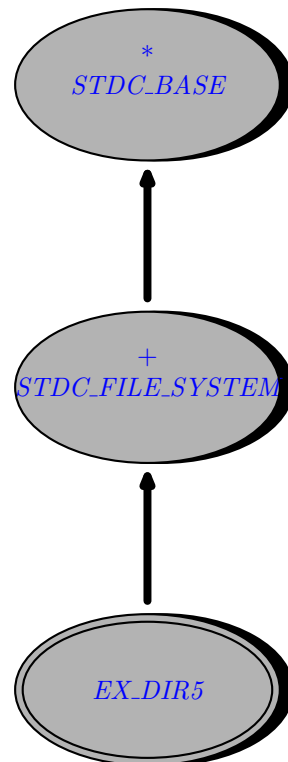


Figure 8.1 BON diagram of deleting and renaming files with Standard C.

feature

```

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

create

    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

create

```

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

create

    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

create

    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

create

    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

end

```

In this chapter:

9.1 Introduction
9.2 Executing a child command
9.3 Reading stdout of a child process
9.4 Catching a signal with Standard C
9.5 Catching a signal with POSIX
9.6 General wait for child handler
9.7 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

create

    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

end
```

9.3 Reading stdout of a child process

It is possible to read the standard output of a child process or write to its standard input. This is one of the easiest ways to communicate with child processes. The `EPX_EXEC_PROCESS` class makes this possible both under Windows and Unix. For example the

creation feature `make _capture _output` makes the standard output of the child available, while `make _capture _input` makes the standard input available.

```

class EX_EXEC1

inherit

    EPX_CURRENT_PROCESS

create

    make

feature

    make is
    local
        ls: EPX_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.fd_stdout.read_string (512)
    until
        ls.fd_stdout.end_of_input
    loop
        print (ls.fd_stdout.last_string)
        ls.fd_stdout.read_string (512)
    end

    -- close captured io
    ls.fd_stdout.close

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

end

```

The three features that give access to the child's standard input, standard output and standard error pipes are named `fd_stdin`, `fd_stdout` and `fd_stderr`.

It is important to wait for the child that has been executed at some point in time, just like any POSIX application 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 only after the `EPX_EXEC_PROCESS.wait_for` command is the exit code of the process available.

It is possible to write to standard input and read standard output and standard error at the same time, but this requires extreme care. It usually leads to code that deadlocks, because the parent process is reading the standard output of the child and the child is waiting for the parent to write to its standard input. Or the child is blocked while writing to its standard output, because its output buffer is full. But the parent process isn't reading the child's standard output, because it is trying to write to the child's standard input.

Under POSIX it is possible to use the buffered features `stdin`, `stdout` and `stderr`. The following example is the same as the previous example, but uses the `POSIX_EXEC_PROCESS` class:

```
class EX_EXEC2

inherit

    POSIX_CURRENT_PROCESS

create

    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.end_of_input
    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
```

It is possible to check if a child process has terminated or not. Pass `False` to the `suspend` parameter of the `EPX_EXEC_PROCESS.wait_for` feature and check `is_terminated` to see if the child process has stopped or not.

9.4 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

create

    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 9.5](#).

9.5 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_SIGNAL1

inherit

    POSIX_CURRENT_PROCESS

    POSIX_CONSTANTS

    POSIX_SIGNAL_HANDLER

create

    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
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.

9.6 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 (Xxxxxxxxxx, 0000)):

```

class EX_SIGNAL2

inherit

    POSIX_CURRENT_PROCESS

    POSIX_CONSTANTS

    POSIX_SIGNAL_HANDLER

create

    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.

9.7 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.

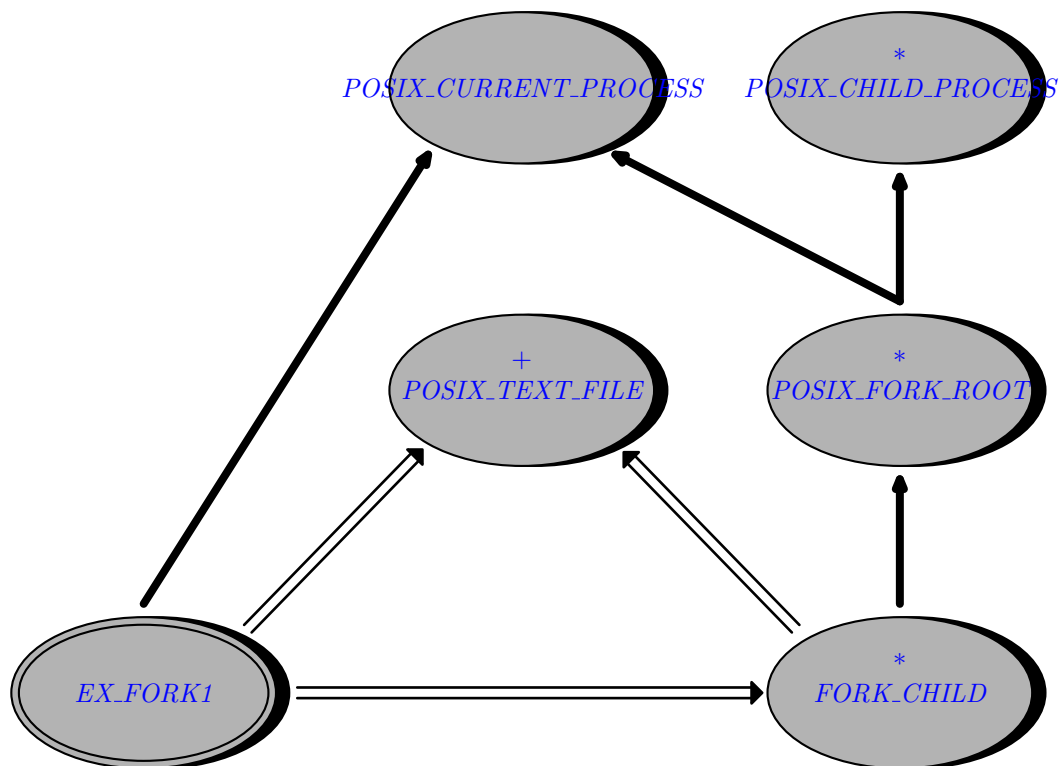


Figure 9.1 BON diagram of forking a child process.

The following class shows the process that forks the child.

```

class

```

```

    EX_FORK1

```

inherit

POSIX_CURRENT_PROCESS

POSIX_FILE_SYSTEM

create

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
```

In this chapter:

10.1*Current time*

10.2*Accessing environment variables*

10.3*Capabilities*

10

Querying the operat- ing system

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
```

```
  create
```

```
    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

  create

    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_ENV1

create

  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_VAR``set_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:

11.1*MIME parsing*

11.2*Sockets*

11.3*Echo client*

11.4*Echo client and server*

11

Working with the network

11.1 *MIME parsing*

Many of the Internet's protocols send data in MIME format. e-POSIX offers a MIME parser in [EPX _MIME _PARSER](#) to parse such data and MIME message creation in [EPX _MIME _PART](#).

MIME messages consist of two parts: a header and a body. The body itself can consist of another header and body. Some examples of using this class are shown in [section 12.7](#).

11.2 *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.

11.3 *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

create

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

  create

    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
end

```

```

    end
  end

end

```

11.4 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

create

  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
  end
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

create

    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)
end

```



```
    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
```

12

Working with the network: ad- vanced top- ics

In this chapter:

12.1 *Introduction*
12.2 *FTP client*
12.3 *HTTP client*
12.4 *HTTP server*
12.5 *IMAP4 client*
12.6 *IRC client*
12.7 *SMTP client*
12.8 *LDIF parser*

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

create

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
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
```

```
create
```

```
make
```

```
feature
```

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

```
make is
```

```
local
```

```
uri: UT_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 [UT_URI](#) class to parse an URI into its components.

12.4 HTTP server

e-POSIX offers a basic HTTP server in [EPX_HTTP_SERVER](#). The following example demonstrates starting such a server and let it listen on the local interface.

```
class EX_HTTP_SERVER1
```

```
inherit
```

```
EPX_CURRENT_PROCESS
```

```
create
```

```
make
```

```
feature
```

```
make is
```

```
local
```

```
server: EPX_HTTP_SERVER
```

```
do
```

```
create server.make (port_to_listen_on, document_root)
```

```
server.set_serve_xhtml_if_supported (False)
```

```
server.listen_locally
```

```
from
```

```
until
```

```
False
```

```
loop
```

```

    server.process_next_requests
    millisleep (100)
end
end

port_to_listen_on: INTEGER is 5566

document_root: STRING is "/var/www/html"

end

```

`EPX_HTTP_SERVER` will say to clients that it serves XHTML instead of HTML. Or in MIME types: `application/xhtml+xml` instead of `text/html`. In case that the HTML pages which are served are not actually XHTML, you will need to turn this option off with a call to `set_serve_xhtml_if_supported`.

In the main loop all available requests are served after which a brief sleep follows. Without the sleep the process would use 100% CPU.

The server will return the files under `/var/www/html` from the file system to the browser. It's also possible to create and register servlets which can respond to requests. A servlet is like a built-in CGI program. A servlet allows maximum control over the response send to the browser, not only the response header, but also the response code send to the client.

A servlet is built after REST principles. A servlet is designed to behave like a resource. You can bind it to a URL and after that it can handle any of the HTTP commands as GET, POST, or PUT that are send to it. By default a servlet will return error code 405, meaning "Method not allowed". The simplest servlet, which always returns 405 is therefore the following:

```

class EX_HTTP_SERVLET1

inherit

    EPX_HTTP_SERVLET

create

    make

end

```

This servlet has to be registered with the HTTP server. The following example shows a virtual HTTP server, one that doesn't have a document root and therefore will never read the file system. It attaches the servlet to the url `/customers`.

```

class EX_HTTP_SERVER2

inherit

    EPX_CURRENT_PROCESS

create

```

```
make
```

```
feature
```

```
make is
```

```
local
```

```
server: EPX_HTTP_SERVER
```

```
servlet: EX_HTTP_SERVLET2
```

```
do
```

```
create server.make_virtual (port_to_listen_on)
```

```
create servlet.make
```

```
server.register_fixed_resource ("/customers", servlet)
```

```
server.listen_locally
```

```
from
```

```
until
```

```
False
```

```
loop
```

```
server.process_next_requests
```

```
millisleep (100)
```

```
end
```

```
end
```

```
port_to_listen_on: INTEGER is 5566
```

```
end
```

You might have noticed it attached servlet `EX_HTTP_SERVLET2`. This servlet is shown below:

```
class EX_HTTP_SERVLET2
```

```
inherit
```

```
EPX_HTTP_SERVLET
```

```
redefine
```

```
get_header
```

```
end
```

```
create
```

```
make
```

```
feature {EPX_HTTP_SERVER} -- Execution
```

```
get_header is
```

```
do
```

```
doctype
```

```
b_html
```

```
b_head
```

```
title ("Customers")
```

```

e_head
b_body
p ("1. John")
p ("2. Luke")
p ("3. Matthew")
p ("4. Pete")
e_body
e_html
write_default_header
add_content_length
end

end

```

Only the `EX_HTTP_SERVLET.get_header` method needs to be overwritten. The format is usually to write the body first and write the header last. This might seem counter-intuitive, but for persistent connections you need to supply a Content-Length if you write a body. Another solution would be to use the chunked transfer encoding, but that isn't explicitly supported yet, so you have to do the work yourself here.

So for dynamically created content, you usually write the body in the header, so you can setup the header. There is also a `EX_HTTP_SERVLET.get_body`, but it is usually not overridden for dynamic content.

The `EPX_HTTP_SERVER` class is responsible for sending the header and the body and to guard against any errors.

In the same manner you can write code to react to PUT, POST or DELETE requests. As browsers usually do not support PUT or DELETE requests, `EPX_HTTP_CONNECTION` will turn a POST request into a PUT or DELETE when it finds a special value. The implementation is in `remap_http_method`. This happens under the following circumstances:

1. The request is a POST request.
2. The POST request is a submit of form fields (regardless of the chosen encoding).
3. There is a form field that starts with the name "http-method:".

In these cases the substring after "http-method:" is taken to override the POST request into whatever is present as substring.

Figure 12.1 shows the BON diagram of the `EPX_HTTP_SERVER`. A server can have zero or more registered servlets and zero or more open connections.



Figure 12.1 BON diagram of `EPX_HTTP_SERVER`.

The server supports persistent connections. In HTTP/1.1 connections are persistent by default. If not requested otherwise, the server will keep the connection open and monitor it to see if any data is coming in. If no data has been sent in the last 15 seconds, the connection is forcibly closed.

The server can have zero or more servlets registered. A single servlet can be connected to multiple URLs by calling `EPX_HTTP_SERVER.register_fixed_resource` with the same servlet.

There is also a `register_dynamic_resource` call to register servlets where part of the data is present in the URL. For example the URL `/customer/1` looks much better than `/customer?id=1`. Register a servlet that takes part of the URL as input as follows:

```
server.register_dynamic_resource ("/customer/(id)", servlet)
```

Every name present between parentheses in such a path is appended to `EPX_HTTP_CONNECTION.request_form_fields`. To a servlet it does therefore not matter if a query is used to input the data, if it is part of a POST or if it was part of the URL. It all becomes input data.

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_IMAP41

inherit

    POSIX_CURRENT_PROCESS

create

    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, `EPX_IRC_CLIENT`. The following example demonstrates logging on to the `#eiffel` channel on `irc.freenode.net` and printing all the messages.

```

class EX_IRC1

create

    make

feature

    make is
    local
        irc: EPX_IRC_CLIENT
        eiffel: EPX_IRC_CHANNEL
    do
        create irc.make (host, username, password)
        irc.set_print_response (True)
        irc.set_real_name ("EiffelBot")
        irc.open
    end
end

```

```

if irc.is_open then
  irc.read_all
  irc.join ("eiffel")
  eiffel := irc.last_joined_channel
  irc.set_blocking_io (True)
from
  irc.read
until
  False
loop
  irc.read
end
  -- We wont come here.,,
  irc.close
end
end

host: STRING is "irc.freenode.net"

username: STRING is "eiffelbot"

password: STRING
  -- n/a

end

```

The printing is done by calling `EPX_IRC_CLIENT.set_print_response`. Not something you probably will use except when debugging. Also we set `set_blocking_io` to `True`, but real IRC clients will be non-blocking.

Look at the test class `TEST_IRC_CLIENT` for more examples, or download the Eiffel Bot from the e-POSIX 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.

12.7.1 Sending plain text email

The following example demonstrates sending a plain text email with this class:

```

class EX_SMTP1

  create

    make

  feature

```

```

make is
local
  message: EPX_MIME_EMAIL
  mail: EPX_SMTP_MAIL
  smtp: EPX_SMTP_CLIENT
do
  create message.make
  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.create_singlepart_body
  message.text_body.append_string ("Hello!")
  create mail.make (sender_mailbox, recipient_mailbox, message)
  create smtp.make (smtp_server_name)
  smtp.open
  smtp.ehlo (my_domain)
  smtp.mail (mail)
  smtp.quit
  smtp.close
end

my_domain: STRING is "example.com"

smtp_server_name: STRING is "localhost"

sender_mailbox: STRING is "berend"

recipient_mailbox: STRING is "berend"

end

```

There are three steps in creating an email:

1. Create the message using `EPX_MIME_EMAIL`, which basically is an `EPX_MIME_PART`. It has and has several convenience routines to quickly create such a message.
2. Create the mail using `EX_SMTP_MAIL`. This class is a container for the sender, the recipients and the actual message that is to be sent.
3. Create an instance of the `EPX_SMTP_CLIENT` class. 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 the Eclass`EPX_SMTP_MAIL` instance created in the previous step.

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. e-POSIX does not support this at the moment. However, passing the local SMTP server is usually sufficient as this server knows how to figure this out.

12.7.2 Sending HTML email

The following example demonstrates sending an HTML text email with this class:

```

class EX_SMTP2

create

    make

feature

    make is
    local
        type_names: expanded EPX_MIME_TYPE_NAMES
        message: EPX_MIME_EMAIL
        mail: EPX_SMTP_MAIL
        smtp: EPX_SMTP_CLIENT
    do
        create message.make
        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_SMTP2")
        message.header.set_content_type_text_html_utf8
        message.create_singlepart_body
        message.text_body.append_string (html)
        create mail.make (sender_mailbox, recipient_mailbox, message)
        create smtp.make (smtp_server_name)
        smtp.open
        smtp.ehlo (my_domain)
        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"

    html: STRING is "[
    <html>
    <head>
        <title>EX_SMTP2</title>
    </head>
    <body>
        <h1>Hello</h1>

```

```

    <p>HTML email, brought to you by eposix.</p>
  </body>
]"

```

end

The main difference is setting the content type to be “text/html”. And the body must be HTML of course.

12.7.3 Sending both text and HTML email

As not all email clients can display HTML, most mailers send both a text and an HTML version. The following example demonstrates how this can be done in e-POSIX:

```

class EX_SMTP3

  create

    make

  feature

    make is
    local
      type_names: expanded EPX_MIME_TYPE_NAMES
      message: EPX_MIME_EMAIL
      ct: EPX_MIME_FIELD_CONTENT_TYPE
      text_part,
      html_part: EPX_MIME_PART
      mail: EPX_SMTP_MAIL
      smtp: EPX_SMTP_CLIENT
    do
      create message.make
      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_SMTP3")
      create ct.make_multipart (
        type_names.mime_subtype_alternative,
        "----=_my-boundary----")
      message.header.add_field (ct)
      message.create_multipart_body

      text_part := message.multipart_body.new_part
      text_part.header.set_content_type (
        type_names.mime_type_text, type_names.mime_subtype_plain,
        "ISO-8859-1")
      text_part.create_singlepart_body
      text_part.text_body.append_string (text)

      html_part := message.multipart_body.new_part
      html_part.header.set_content_type (

```

```

    type_names.mime_type_text, type_names.mime_subtype_html,
    "ISO-8859-1")
    html_part.create_singlepart_body
    html_part.text_body.append_string (html)

    create mail.make (sender_mailbox, recipient_mailbox, message)
    create smtp.make (smtp_server_name)
    smtp.open
    smtp.ehlo (my_domain)
    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"

html: STRING is "[
<html>
<head>
  <title>EX_SSMTP3</title>
</head>
<body>
  <h1>Hello</h1>
  <p>HTML email, brought to you by eposix.</p>
</body>
]"

text: STRING is "Hello%N%NHTML email, brought to you by eposix."

end

```

We set the content type to be “multipart/alternative”, and create two parts. The first part is content type “text/plain” and the second is the content type “text/html”.

12.7.4 Sending attachments

Multipart emails are also the key to sending attachments. The following example demonstrates how this can be done in e-POSIX by attaching the example itself:

```

class EX_SSMTP4

inherit

    EPX_FILE_SYSTEM

```

create

make

feature

make is

local

message: EPX_MIME_EMAIL

ct: EPX_MIME_FIELD_CONTENT_TYPE

text_part,

file_part: EPX_MIME_PART

do

create *message.make*

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_SMTp4")

create *ct.make_multipart* (

type_names.mime_subtype_mixed,

"----=_my-boundary----")

message.header.add_field (ct)

message.create_multipart_body

text_part := message.multipart_body.new_part

text_part.header.set_content_type (

type_names.mime_type_text, type_names.mime_subtype_plain,

"ISO-8859-1")

text_part.create_singlepart_body

text_part.text_body.append_string ("Here is the file.")

file_part := message.multipart_body.new_part

file_part.header.set_content_type (

type_names.mime_type_text, type_names.mime_subtype_plain, Void)

file_part.header.content_type.set_parameter ("name", filename)

file_part.create_singlepart_body

file_part.text_body.append_string (file_content_as_string (filename))

send_message (message)

end

send_message (a_message: EPX_MIME_EMAIL) **is**

local

mail: EPX_SMTp_MAIL

smtp: EPX_SMTp_CLIENT

do

create *mail.make* (sender_mailbox, recipient_mailbox, a_message)

create *smtp.make* (smtp_server_name)

smtp.open

smtp.ehlo (my_domain)

```

smtp.mail (mail)
smtp.quit
smtp.close
end

my_domain: STRING is "example.com"

smtp_server_name: STRING is "localhost"

sender_mailbox: STRING is "berend"

recipient_mailbox: STRING is "berend@bmach"

type_names: EPX_MIME_TYPE_NAMES is
do
  create Result
end

filename: STRING is "ex_smtp4.e"

end

```

The attachment in this case has to be a text file. Anything that has binary data, i.e. characters lower than character code 32, needs to be encoded first. The following extract demonstrates this encoding:

```

file_part := message.multipart_body.new_part
file_part.header.set_content_type (
  type_names.mime_type_application, type_names.mime_subtype_pdf, Void)
file_part.header.content_type.set_parameter ("name", filename)
file_part.create_base64_body
file_part.text_body.append_string (file_content_as_string (filename))

```

All the other code is just the same as in the previous example, just the file part is different.

An even shorter method to add an attachment is by using `EPX_MIME_EMAIL.attach_file` which basically does what the above lines of code do.

12.8 LDIF parser

e-POSIX contains an LDIF (LDAP Data Interchange Format) parser, see RFC 2849.

In this chapter:

13.1*Introduction*
13.2*Windows*
13.3*Creating a daemon*
13.4*Logging messages and errors*
13.5*ULM 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
```

```
create
```

```
    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

create

  make

feature

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

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

```

```

        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-ulm-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

create

make

feature -- Initialization

make is
local
    logger: NET_LOGGER
    handler: EPX_LOG_HANDLER
    field: NET_LOGGER_FIELD
    fields: DS_LINKED_LIST [NET_LOGGER_FIELD]
do
    -- Create handler and logger
    create handler.make (identification)
    create logger.make (handler, system_name)

```

```

-- Log a simple message
logger.write_msg (logger.levels.warning, "testing", "Hello World.")

-- Log a message with a custom field
create fields.make
create field.make ("myField", "127.0.0.1")
fields.put (field, 0)
logger.write (logger.levels.info, "testing", fields)
end

feature -- Access

identification: STRING is "example"

system_name: STRING is "ex_ulm"

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.

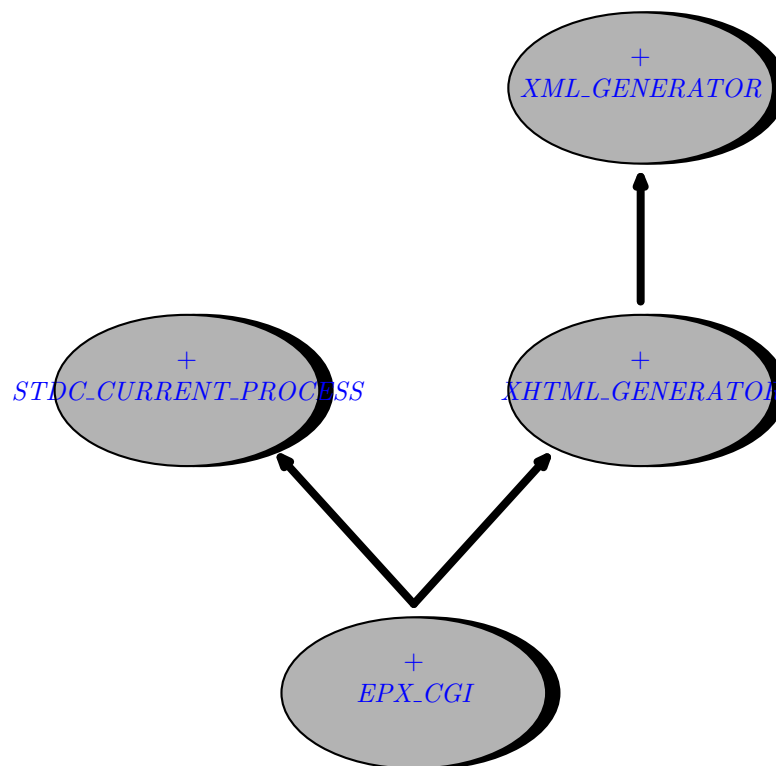


Figure 14.1 BON diagram of `EPX_CGI`.

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_CGI1
```

```
inherit
```

```
    EPX_CGI
```

```
create
```

```
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

create

    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

create

```

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.end_of_input
            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

create

    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:

15.1 Error handling with exceptions
15.2 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 (XXXXXXXXXX, 0000) 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 hold 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 (XXXXXXXXXX, 0000)). 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 (XXXXXXXXXX, 0000)) 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_ERROR1
```

```
inherit
```

```

    POSIX_CURRENT_PROCESS

create

    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 (XXXXXXXXXX, 0000)). 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 usefull 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 safes 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

create

  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
```

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
```

```
create
```

```
    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:

16.1*Denial of service attacks*
16.2*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:

17.1 Making C Headers available to Eiffel
*17.2 Distinction between Standard C and
POSIX headers*
17.3 C 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
```

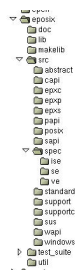


Figure 17.1 e-POSIX
directory structure

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](#).

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

Table 17.1 e-POSIX prefix conventions

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.

In this chapter:

A Posix func- tion to Eiffel class map- ping 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	Comments
abort	<code><stdlib.h></code>	<code>STDC_CURRENT_PROCESS.abort</code>	
accept	<code><sys/socket.h></code>	<code>EPX_TCP_SERVER_SOCKET.accept</code>	
access	<code><unistd.h></code>	<code>ABSTRACT_FILE_SYSTEM.is_accessible</code>	
aio_cancel	<code><aio.h></code>	<code>POSIX_ASYNC_IO_REQUEST.cancel</code>	
aio_error	<code><aio.h></code>	<code>POSIX_ASYNC_IO_REQUEST.is_pending</code>	
aio_fsync	<code><aio.h></code>	<code>POSIX_ASYNC_IO_REQUEST.synchronize</code>	
aio_read	<code><aio.h></code>	<code>POSIX_ASYNC_IO_REQUEST.read</code>	
aio_return	<code><aio.h></code>	<code>POSIX_ASYNC_IO_REQUEST.return_status</code>	
aio_suspend	<code><aio.h></code>	<code>POSIX_ASYNC_IO_REQUEST.wait_for</code>	
aio_write	<code><aio.h></code>	<code>POSIX_ASYNC_IO_REQUEST.write</code>	
alarm	<code><unistd.h></code>	<code>POSIX_TIMED_COMMAND</code>	
asctime	<code><time.h></code>	<code>STDC_TIME.default_format</code>	
atexit	<code><stdlib.h></code>	<code>STDC_EXIT_SWITCH.install</code>	Use <code>STD... _ACCESS... to acces...</code>
bind	<code><sys/socket.h></code>	<code>EPX_TCP_SERVER_SOCKET.listen_by_address</code>	
calloc	<code><stdlib.h></code>	<code>STDC_BUFFER.allocate_and_clear</code>	
cfgetispeed	<code><termios.h></code>	<code>POSIX_TERMIOS.input_speed</code>	
cfgetospeed	<code><termios.h></code>	<code>POSIX_TERMIOS.output_speed</code>	
cfsetispeed	<code><termios.h></code>	<code>POSIX_TERMIOS.set_input_speed</code>	
cfsetospeed	<code><termios.h></code>	<code>POSIX_TERMIOS.set_output_speed</code>	
chdir	<code><unistd.h></code>	<code>POSIX_FILE_SYSTEM.change_directory</code>	
chmod	<code><sys/stat.h></code>	<code>POSIX_FILE_SYSTEM.change_mode</code>	
chown	<code><unistd.h></code>	<code>POSIX_PERMISSIONS_PATH.apply_owner_and_group</code>	
clearerr	<code><stdio.h></code>	<code>STDC_FILE.clear_error</code>	
clock	<code><time.h></code>	<code>STDC_CURRENT_PROCESS.clock</code>	
clock_getcpuclockid	<code><time.h></code>		
clock_getres	<code><time.h></code>	<code>SUS_SYSTEM.real_time_clock_resolution</code>	
clock_gettime	<code><time.h></code>	<code>SUS_SYSTEM.real_time_clock</code>	
clock_nanosleep	<code><time.h></code>		
clock_settime	<code><time.h></code>		
close	<code><unistd.h></code>	<code>EPX_FILE_DESCRIPTOR.close</code>	

closedir	<code><dirent.h></code>	POSIX_DIRECTORY
closelog	<code><syslog.h></code>	SUS_SYSLOG.close
confstr	<code><unistd.h></code>	
connect	<code><sys/socket.h></code>	EPX_TCP_CLIENT_SOCKET.open_by_address, open_by_name_and_port
creat	<code><fcntl.h></code>	EPX_FILE_DESCRIPTOR.create_read_write
ctermid	<code><unistd.h></code>	
ctime	<code><time.h></code>	
cuserid	<code><stdio.h></code>	
daylight	<code><time.h></code>	
difftime	<code><time.h></code>	STDC_TIME
dup	<code><unistd.h></code>	EPX_FILE_DESCRIPTOR.make_as_duplicate
dup2	<code><unistd.h></code>	EPX_FILE_DESCRIPTOR.make_as_duplicate
endgrent	<code><grp.h></code>	
endhostent	<code><netdb.h></code>	
endnetent	<code><netdb.h></code>	
endprotoent	<code><netdb.h></code>	
endpwent	<code><pwd.h></code>	
endservent	<code><netdb.h></code>	
execl	<code><unistd.h></code>	
execle	<code><unistd.h></code>	
execlp	<code><unistd.h></code>	
execv	<code><unistd.h></code>	
execve	<code><unistd.h></code>	
execvp	<code><unistd.h></code>	EPX_EXEC_PROCESS.execute
exit	<code><stdlib.h></code>	STDC_CURRENT_PROCESS.exit
_exit	<code><unistd.h></code>	
fchmod	<code><sys/stat.h></code>	
fchown	<code><sys/stat.h></code>	
fclose	<code><stdio.h></code>	STDC_FILE.close
fcntl	<code><unistd.h></code>	POSIX_FILE_DESCRIPTOR
fdatasync	<code><unistd.h></code>	POSIX_FILE_DESCRIPTOR.synchronize_data
fdopen	<code><stdio.h></code>	POSIX_FILE.make_from_file_descriptor
feof	<code><stdio.h></code>	STDC_FILE.eof
ferror	<code><stdio.h></code>	STDC_FILE.error
fflush	<code><stdio.h></code>	STDC_FILE.flush
fgetc	<code><stdio.h></code>	STDC_FILE.get_character
fgetpos	<code><stdio.h></code>	STDC_FILE.get_position
fgets	<code><stdio.h></code>	STDC_FILE.get_string
fileno	<code><stdio.h></code>	POSIX_FILE_DESCRIPTOR.make_from_file
flockfile	<code><stdio.h></code>	
fopen	<code><stdio.h></code>	STDC_FILE
fork	<code><unistd.h></code>	POSIX_CURRENT_PROCESS.fork
fpathconf	<code><unistd.h></code>	
fprintf	<code><stdio.h></code>	
fputc	<code><stdio.h></code>	STDC_FILE.putc
fputs	<code><stdio.h></code>	STDC_FILE.put_string
fread	<code><stdio.h></code>	STDC_FILE.read
free	<code><stdlib.h></code>	STDC_BUFFER.deallocate
freopen	<code><stdio.h></code>	STDC_FILE.reopen

Can be
STDC_
see get

See exe
See exe
See exe
See exe
See exe

attempt
set _l
This fu
able on
POSIX s
cases i
fsync.

various
features

not app

Also re
read _c

fseek	\$<\$stdio.h\$>\$	STDC_FILE. seek	Also see and see
fsetpos	\$<\$stdio.h\$>\$	STDC_FILE. set_position	
fstat	\$<\$sys/stat.h\$>\$	POSIX_STATUS	Return _DESCR.
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\$>\$		See fgetc
getchar	\$<\$stdio.h\$>\$		See fgetc
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\$>\$		See fgetc
gettimeofday	\$<\$sys/time.h\$>\$		SUS_T
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	
ioctl	\$<\$stropts.h\$>\$	SAPI_STROPTS. posix_ioctl	
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	Also see and see
malloc	\$<\$stdlib.h\$>\$	STDC_BUFFER. allocate	
memcpy	\$<\$string.h\$>\$	STDC_BUFFER. memory_copy	See also
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	
mkstemp	\$<\$stdlib.h\$>\$	SUS_TEMPORARY_FILE. make	
mktime	\$<\$time.h\$>\$	STDC_TIME. set_date_time	Also see _time.
mlockall	\$<\$sys/mman.h\$>\$		

mlock	<code><sys/mman.h></code>	
mmap	<code><sys/mman.h></code>	POSIX_MEMORY_MAP
mprotect	<code><sys/mman.h></code>	
mq_receive	<code><mqqueue.h></code>	
mq_close	<code><mqqueue.h></code>	
mq_getattr	<code><mqqueue.h></code>	
mq_notify	<code><mqqueue.h></code>	
mq_open	<code><mqqueue.h></code>	
mq_send	<code><mqqueue.h></code>	
mq_setattr	<code><mqqueue.h></code>	
mq_unlink	<code><mqqueue.h></code>	
msync	<code><sys/mman.h></code>	
munlockall	<code><sys/mman.h></code>	
munlock	<code><sys/mman.h></code>	
munmap	<code><sys/mman.h></code>	POSIX_MEMORY_MAP
nanosleep	<code><time.h></code>	SUS_CURRENT_PROCESS.nanosleep
ntohl	<code><netinet/in.h></code>	SAPI_IN.posix_ntohl
ntohs	<code><netinet/in.h></code>	SAPI_IN.posix_ntohs
open	<code><fcntl.h></code>	EPX_FILE_DESCRIPTOR.open
opendir	<code><dirent.h></code>	POSIX_DIRECTORY
openlog	<code><syslog.h></code>	SUS_SYSLOG.open
pathconf	<code><unistd.h></code>	POSIX_DIRECTORY.max_filename_length
pause	<code><unistd.h></code>	EPX_CURRENT_PROCESS.pause
perror	<code><stdio.h></code>	
pipe	<code><unistd.h></code>	EPX_PIPE.make
printf	<code><stdio.h></code>	
putc	<code><stdio.h></code>	
putchar	<code><stdio.h></code>	
puts	<code><stdio.h></code>	
raise	<code><signal.h></code>	STDC_SIGNAL.raise
rand	<code><stdlib.h></code>	STDC_CURRENT_PROCESS.random
read	<code><unistd.h></code>	EPX_FILE_DESCRIPTOR.read
readdir	<code><dirent.h></code>	POSIX_DIRECTORY
realloc	<code><stdlib.h></code>	STDC_BUFFER.resize
remove	<code><stdio.h></code>	POSIX_FILE_SYSTEM.remove_file
rename	<code><unistd.h></code>	POSIX_FILE_SYSTEM.rename_to
rewind	<code><stdio.h></code>	STDC_FILE.rewind
rewinddir	<code><dirent.h></code>	POSIX_DIRECTORY
rmdir	<code><unistd.h></code>	EPX_FILE_SYSTEM.remove_directory
scanf	<code><stdio.h></code>	
select	<code><sys/select.h></code>	EPX_SELECT
sem_close	<code><semaphore.h></code>	
sem_destroy	<code><semaphore.h></code>	
sem_getvalue	<code><semaphore.h></code>	
sem_init	<code><semaphore.h></code>	POSIX_UNNAMED_SEMAPHORE.create_shared
sem_open	<code><semaphore.h></code>	
sem_post	<code><semaphore.h></code>	POSIX_SEMAPHORE.release
sem_trywait	<code><semaphore.h></code>	POSIX_SEMAPHORE.attempt_acquire
sem_unlink	<code><semaphore.h></code>	
sem_wait	<code><semaphore.h></code>	POSIX_SEMAPHORE.acquire
setbuf	<code><stdio.h></code>	STDC_FILE.set_buffer
setgid	<code><unistd.h></code>	POSIX_CURRENT_PROCESS.set_group_id
setlocale	<code><locale.h></code>	STDC_CURRENT_PROCESS.set_locale
setpgid	<code><unistd.h></code>	PAPI_UNISTD.posix_setsid

Also op
read
_write

e-POSIX
ception

not app
See fpu
See fpu
See fpu

not app

And cr

Also re
Also se
and set

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\$>\$	

Also re
Also se
and set
And cr
_read,

Also re
_signal
_signal

Not ap

Not ap

Use fea
or erro

Also ap
apply

Various

See also
Not ap
Not ap

<code>vsprintf</code>	<code><stdio.h></code>	
<code>wait</code>	<code><sys/wait.h></code>	<code>POSIX _CURRENT _PROCESS. wait</code>
<code>waitpid</code>	<code><sys/wait.h></code>	<code>POSIX _FORK _ROOT. wait _pid</code>
<code>write</code>	<code><unistd.h></code>	<code>EPX _FILE _DESCRIPTOR. write</code>

Not ap

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 file-names 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

Index

[23, 68
) 54
abort
 STDC_CURRENT_PROCESS 96
accept
 EPX_TCP_SERVER_SOCKET 96
acquire
 POSIX_SEMAPHORE 99
add
 POSIX_SIGNAL_SET 100
add_data
 EPX_CGI 80
add_raw
 EPX_CGI 81
add_to_blocked_signals
 POSIX_SIGNAL_SET 100
allocate
 STDC_BUFFER 98
allocate_and_clear
 STDC_BUFFER 17, 96
apply
 POSIX_SIGNAL 46
apply_drain
 POSIX_TERMIOS 100
apply_flush
 POSIX_TERMIOS 100
apply_now
 POSIX_TERMIOS 100
apply_owner_and_group
 POSIX_PERMISSIONS_PATH 96
assert_key_value_pairs_created
 EPX_CGI 84
attach_file
 EPX_MIME_EMAIL 73
attempt_acquire
 POSIX_SEMAPHORE 99
attempt_lock
 POSIX_FILE_DESCRIPTOR 97
attempt_open_read
 POSIX_TEXT_FILE 86
b_a
 EPX_CGI 81
b_form_get
 EPX_CGI 82
b_form_post
 EPX_CGI 82
b_input
 EPX_CGI 82
b_p
 EPX_CGI 81
browse_directory
 POSIX_FILE_SYSTEM 40, 87
cancel
 POSIX_ASYNC_IO_REQUEST 96
change_directory
 POSIX_FILE_SYSTEM 96
change_mode
 POSIX_FILE_SYSTEM 96
clear_error
 STDC_FILE 96
clear_first
 STDC_ERRNO 89
clock
 STDC_CURRENT_PROCESS 96
close
 EPX_FILE_DESCRIPTOR 96
 EPX_SMTP_CLIENT 68
 STDC_FILE 92, 97
 SUS_SYSLOG 97
content_text_html
 EPX_CGI 84
copy_from
 STDC_BUFFER 98
create_fifo
 POSIX_FILE_SYSTEM 14, 98
create_read_write
 EPX_FILE_DESCRIPTOR 97
create_shared
 POSIX_UNNAMED_SEMAPHORE
 99
create_unshared
 POSIX_UNNAMED_SEMAPHORE
 99
create_write
 POSIX_SHARED_MEMORY 100
current_directory
 POSIX_FILE_SYSTEM 98
deallocate
 STDC_BUFFER 97

default_format
 POSIX_TIME 51
 STDC_TIME 96

detach
 POSIX_DAEMON 74

dispose
 MEMORY 92

doctype
 EPX_XML_WRITER 80

doctype_transitional
 EPX_XML_WRITER 80

effective_group_id
 POSIX_CURRENT_PROCESS 98

effective_user_id
 POSIX_CURRENT_PROCESS 98

ehlo
 EPX_SMTP_CLIENT 68

end_of_input
 POSIX_TEXT_FILE 24

eof
 STDC_FILE 97

_errno 5

errno
 POSIX_FILE_DESCRIPTOR 89

errno.first_value
 POSIX_FILE_DESCRIPTOR 89

errno.value
 POSIX_FILE_DESCRIPTOR 89

error
 STDC_FILE 97
 SUS_SYSLOG 100

execute
 EPX_CGI 79, 80
 EPX_EXEC_PROCESS 97
 POSIX_DAEMON 74
 POSIX_FORK_ROOT 48
 POSIX_SHELL_COMMAND 42

_exit 97

exit
 STDC_CURRENT_PROCESS 97

exit_switch
 STDC_EXIT_SWITCH_ACCESSOR 96

expand_path
 POSIX_FILE_SYSTEM 25

fd_stderr
 EPX_EXEC_PROCESS 43

fd_stdin
 EPX_CURRENT_PROCESS 31
 EPX_EXEC_PROCESS 43

fd_stdout
 EPX_CURRENT_PROCESS 31
 EPX_EXEC_PROCESS 43

file
 EPX_KEYVALUE 84

fill_with
 STDC_BUFFER 98

first_value
 POSIX_FILE_DESCRIPTOR 89
 STDC_ERRNO 89

flush
 STDC_FILE 97

flush_input
 POSIX_TERMIOS 100

force_remove_directory
 ABSTRACT_FILE_SYSTEM 104

fork
 POSIX_CURRENT_PROCESS 48, 97

format
 POSIX_TIME 51
 STDC_TIME 100

forth_recursive
 ABSTRACT_DIRECTORY 102

get_body
 EX_HTTP_SERVLET 64

get_character
 STDC_FILE 97

get_header
 EX_HTTP_SERVLET 64

get_lock
 POSIX_FILE_DESCRIPTOR 14, 29, 97

get_position
 POSIX_FILE 25
 STDC_FILE 97

get_string
 STDC_FILE 97

has
 POSIX_SIGNAL_SET 100

input_speed
 POSIX_TERMIOS 96

input_text
 EPX_CGI 82

install
 STDC_EXIT_SWITCH 96

is_accessible
 ABSTRACT_FILE_SYSTEM 96

is_attached_to_terminal
 EPX_FILE_DESCRIPTOR 98

- is_blocking_io*
 - ABSTRACT_FILE_DESCRIPTOR 34
- is_in_group*
 - POSIX_CURRENT_PROCESS 98
- is_modifiable*
 - POSIX_FILE_SYSTEM 38
- is_pending*
 - POSIX_ASYNC_IO_REQUEST 96
- is_readable*
 - POSIX_FILE_SYSTEM 39
- is_terminated*
 - EPX_EXEC_PROCESS 45
- is_terminated_normally*
 - POSIX_FORK_ROOT 104
- kill*
 - POSIX_PROCESS 98
- last_string*
 - POSIX_TEXT_FILE 24
- link*
 - POSIX_FILE_SYSTEM 98
- listen_by_address*
 - EPX_TCP_SERVER_SOCKET 96
- local_date_string*
 - POSIX_TIME 51
- local_time_string*
 - POSIX_TIME 51
- log_event*
 - ULM_LOGGING 77
- login_name*
 - POSIX_CURRENT_PROCESS 98
- log_message*
 - ULM_LOGGING 77
- mail*
 - EPX_SMTP_CLIENT 68
- make*
 - EPX_LOG_HANDLER 78
 - EPX_PIPE 99
 - POSIX_TERMIOS 100
 - STDC_TEMPORARY_FILE 100
 - SUS_TEMPORARY_FILE 98
- make_as_duplicate*
 - EPX_FILE_DESCRIPTOR 97
 - POSIX_FILE_DESCRIPTOR 32
- make_capture_input*
 - EPX_EXEC_PROCESS 43
- make_capture_output*
 - EPX_EXEC_PROCESS 43
- make_directory*
 - EXP_FTP_CLIENT 60
 - POSIX_FILE_SYSTEM 98
- make_empty*
 - POSIX_SIGNAL_SET 100
- make_expand*
 - STDC_PATH 38
- make_from_file*
 - POSIX_FILE_DESCRIPTOR 97
- make_from_file_descriptor*
 - POSIX_FILE 97
- make_from_gid*
 - POSIX_GROUP 98
- make_from_name*
 - POSIX_GROUP 98
 - POSIX_USER 98
- make_from_now*
 - POSIX_TIME 51
- make_from_uid*
 - POSIX_USER 98
- make_from_unix_time*
 - STDC_TIME 100
- make_full*
 - POSIX_SIGNAL_SET 100
- make_pending*
 - POSIX_SIGNAL_SET 100
- max_filename_length*
 - POSIX_DIRECTORY 99
- memory_copy*
 - STDC_BUFFER 98
- memory_move*
 - STDC_BUFFER 98
- nanosleep*
 - SUS_CURRENT_PROCESS 99
- notice*
 - SUS_SYSLOG 100
- open*
 - EPX_FILE_DESCRIPTOR 99
 - POSIX_FILE 8
 - SUS_SYSLOG 99
- open_by_address*
 - EPX_TCP_CLIENT_SOCKET 97
- open_by_name_and_port*
 - EPX_TCP_CLIENT_SOCKET 97
- open_read*
 - EPX_FILE_DESCRIPTOR 99
 - POSIX_FILE 8
 - POSIX_SHARED_MEMORY 100
 - POSIX_TEXT_FILE 86
- open_read_write*
 - EPX_FILE_DESCRIPTOR 99
 - POSIX_SHARED_MEMORY 100

open_write
 EPX_FILE_DESCRIPTOR 99

out
 EPX_IP4_ADDRESS 98

output_speed
 POSIX_TERMIOS 96

parent_pid
 POSIX_CURRENT_PROCESS 98

parse
 STDC_PATH 38

pause
 EPX_CURRENT_PROCESS 99

peek_int16
 STDC_BUFFER 18

peek_int16_big_endian
 STDC_BUFFER 18

peek_int16_little_endian
 STDC_BUFFER 18

peek_int32
 STDC_BUFFER 18

peek_uint16
 STDC_BUFFER 18

permissions
 POSIX_FILE_SYSTEM 39

pid
 POSIX_CURRENT_PROCESS 11, 98

poke_int32_big_endian
 STDC_BUFFER 18

posix_htonl
 SAPI_IN 98

posix_htons
 SAPI_IN 98

posix_ioctl
 SAPI_STROPTS 98

posix_memcmp
 CAPI_STRING 98

posix_ntohl
 SAPI_IN 99

posix_ntohs
 SAPI_IN 99

posix_setsid
 PAPI_UNISTD 99, 100

process_group_id
 POSIX_CURRENT_PROCESS 98

prune
 POSIX_SIGNAL_SET 100

putc
 STDC_FILE 97

puts
 EPX_CGI 81

put_string
 POSIX_FILE_DESCRIPTOR 88, 89
 STDC_FILE 97

quit
 EPX_SMTP_CLIENT 68

raise
 STDC_SIGNAL 99, 100

raise_posix_error
 STDC_BASE 104

random
 STDC_CURRENT_PROCESS 99

raw_value
 EPX_CGI 82

read
 ABSTRACT_FILE_DESCRIPTOR 34
 EPX_FILE_DESCRIPTOR 99
 POSIX_ASYNC_IO_REQUEST 35, 96
 POSIX_FILE 24
 STDC_FILE 97

read_buffer
 POSIX_FILE 24

read_character
 STDC_FILE 97

read_line
 [23
 ABSTRACT_FILE_DESCRIPTOR 30

read_string
 [23
 ABSTRACT_FILE_DESCRIPTOR 21
 POSIX_TEXT_FILE 24
 STDC_FILE 97

real_group_id
 POSIX_CURRENT_PROCESS 98

real_time_clock
 SUS_SYSTEM 96

real_time_clock_resolution
 SUS_SYSTEM 96

real_user_id
 POSIX_CURRENT_PROCESS 98

reestablish
 STDC_SIGNAL_HANDLER 47

refresh
 POSIX_PERMISSIONS 39

- register_dynamic_resource*
 - EPX_HTTP_SERVER 65
- register_fixed_resource*
 - EPX_HTTP_SERVER 65
- release*
 - POSIX_SEMAPHORE 99
- remap_http_method*
 - EPX_HTTP_CONNECTION 64
- remove_directory*
 - EPX_FILE_SYSTEM 99
 - EXP_FTP_CLIENT 60
- remove_file*
 - EXP_FTP_CLIENT 60
 - GENERAL 5
 - POSIX_FILE_SYSTEM 5, 88, 99
- remove_from_blocked_signals*
 - POSIX_SIGNAL_SET 100
- rename_to*
 - EXP_FTP_CLIENT 60
 - POSIX_FILE_SYSTEM 99
- reopen*
 - STDC_FILE 97
- request_form_fields*
 - EPX_HTTP_CONNECTION 65
- resize*
 - STDC_BUFFER 99
- restore_group_id*
 - POSIX_CURRENT_PROCESS 99
- restore_user_id*
 - POSIX_CURRENT_PROCESS 100
- return_status*
 - POSIX_ASYNC_IO_REQUEST 96
- rewind*
 - STDC_FILE 99
- save_uploaded_files*
 - EX_CGI3 84
- security.cpu.check_process_time*
 - STDC_FILE 92
- security.cpu.set_max_process_time*
 - STDC_FILE 92
- security.error_handling.disable_exceptions*
 - STDC_SECURITY_ACCESSOR 88
- security.error_handling.enable_exceptions*
 - STDC_SECURITY_ACCESSOR 88
- security.files.set_max_open_files*
 - STRING 92
- security.memory.set_max_allocation*
 - STDC_SECURITY_ACCESSOR 91
- security.memory.set_max_single_allocation*
 - STRING 92
- seek*
 - EPX_FILE_DESCRIPTOR 98
 - POSIX_FILE 25
 - STDC_FILE 98
- seek_from_current*
 - EPX_FILE_DESCRIPTOR 98
 - STDC_FILE 98
- seek_from_end*
 - EPX_FILE_DESCRIPTOR 98
 - STDC_FILE 98
- set_allow_anyone_read*
 - POSIX_PERMISSIONS 39
- set_allow_group_write*
 - POSIX_PERMISSIONS 39
- set_blocked_signals*
 - POSIX_SIGNAL_SET 100
- set_blocking_io*
 - ABSTRACT_FILE_DESCRIPTOR 34
 - EPX_IRC_CLIENT 67
- set_buffer*
 - POSIX_ASYNC_IO_REQUEST 35
 - STDC_FILE 99
- set_count*
 - POSIX_ASYNC_IO_REQUEST 35
- set_date*
 - STDC_TIME 98
- set_date_time*
 - STDC_TIME 98
- set_full_buffering*
 - STDC_FILE 100
- set_group_id*
 - POSIX_CURRENT_PROCESS 99
- set_handler*
 - POSIX_SIGNAL 46, 47
- set_input_speed*
 - POSIX_TERMIOS 96
- set_line_buffering*
 - STDC_FILE 100
- set_locale*
 - STDC_CURRENT_PROCESS 99
- set_lock*
 - POSIX_FILE_DESCRIPTOR 97
- set_native_locale*
 - STDC_CURRENT_PROCESS 99
- set_native_time*
 - STDC_CURRENT_PROCESS 99
- set_no_buffering*
 - STDC_FILE 100

set_offset
 POSIX_ASYNC_IO_REQUEST 35

set_output_speed
 POSIX_TERMIOS 96

set_position
 POSIX_FILE 25
 STDC_FILE 98

set_print_response
 EPX_IRC_CLIENT 67

set_random_seed
 STDC_CURRENT_PROCESS 100

set_serve_xhtml_if_supported
 EPX_HTTP_SERVER 62

set_time
 STDC_TIME 98

set_user_id
 POSIX_CURRENT_PROCESS 100

signalled
 POSIX_SIGNAL_HANDLER 46, 47

sleep
 EPX_CURRENT_PROCESS 46
 POSIX_CURRENT_PROCESS 100

/src/library.xace 6, 6

status
 EPX_FILE_DESCRIPTOR 98
 POSIX_FILE_DESCRIPTOR 40
 STDC_ERRNO
 POSIX_FILE_DESCRIPTOR 89

stderr
 EPX_EXEC_PROCESS 44

stdin
 EPX_CURRENT_PROCESS 31
 EPX_EXEC_PROCESS 44

stdout
 EPX_CURRENT_PROCESS 31
 EPX_EXEC_PROCESS 44

store
 EXP_FTP_CLIENT 60

supports_nonblocking_io
 ABSTRACT_FILE_DESCRIPTOR 34

SUS_ENV_VAR
 STDC_ENV_VAR 53

suspend
 POSIX_SIGNAL_SET 100

synchronize
 POSIX_ASYNC_IO_REQUEST 35, 96
 POSIX_FILE_DESCRIPTOR 98

synchronize_data
 POSIX_FILE_DESCRIPTOR 97

tell
 POSIX_FILE 25
 STDC_FILE 98

temporary_file_name
 STDC_FILE_SYSTEM 100

to_local
 POSIX_TIME 51
 STDC_TIME 98

touch
 POSIX_FILE_SYSTEM 100

to_utc
 POSIX_TIME 51
 STDC_TIME 98

ttyname
 POSIX_FILE_DESCRIPTOR 100

unfinished_xml
 EPX_XML_WRITER 80

ungetc
 STDC_FILE 100

unlink
 POSIX_FILE_SYSTEM 100

unlink_shared_memory_object
 POSIX_FILE_SYSTEM 100

utime
 POSIX_FILE_SYSTEM 100

value
 EPX_CGI 82, 84
 STDC_ENV_VAR 98

wait
 POSIX_CURRENT_PROCESS 12, 101

waited_child_pid
 POSIX_CURRENT_PROCESS 12

wait_for
 EPX_EXEC_PROCESS 43, 45
 POSIX_ASYNC_IO_REQUEST 35, 96
 POSIX_CHILD 12

wait_pid
 POSIX_FORK_ROOT 101

write
 ABSTRACT_FILE_DESCRIPTOR 34
 EPX_FILE_DESCRIPTOR 101
 POSIX_ASYNC_IO_REQUEST 35, 96
 STDC_FILE 98
 96

a

abort 96
ABSTRACT_FILE_DESCRIPTOR 15,
21, 30, 34
accept 96
access 96, 104
aio_cancel 96
aio_error 96
aio_fsync 96
aio.h 96, 98
aio_read 96
aio_return 96
aio_suspend 96
aio_write 96
alarm 96
ANY 5
arpa/inet.h 98
asctime 96
atexit 96

b

backslash 25
big endian 18
binary file 25
binary mode 30
binary stdin 31
binary stdout 31
bind 96

c

calloc 96
CAPI_STDIO 8, 94
C compiler
 Borland 2
 lcc 2
 Microsoft 2
 mingw 5
cecil.se 5
cfgetispeed 96
cfgetospeed 96
cfsetispeed 96
cfsetospeed 96
cgi 79
 enumerating all values 84
 file upload 81
 redirect 84
chdir 96
chmod 96
chown 96
clearerr 96

clock 96
clock_getcpuclockid 96
clock_getres 96
clock_gettime 96
clock_nanosleep 96
clock_settime 96
close 96
closedir 97
closelog 97
compiler.se 14
configure 2, 6
confstr 97
connect 97
Content-Length 64
creat 97, 103
c_stdio.c 94
c_stdio.h 94
ctermid 97
ctime 97
Ctrl
 C 45, 46
ctype.h 101
cuserid 97
CYGWIN 31
Cygwin 6
cygwin 6, 9
CYGWIN_API 104

d

daylight 97
DELETE request 64
difftime 97
directory
 browse 40
 change 38
 create 38
 remove 38
 test_suite 16
dirent.h 97, 99
dup 97
dup2 97

e

EEXIST 87
eiffel.h 93
elj-win32 2
endgrent 97
endhostent 97
endnetent 97
endprotoent 97

- endpwent 97
- endservent 97
- ENOSYS 14
- environment variable 25
 - CFLAGS 6
 - CYGWIN 31
 - EPOSIX 1
- Environment variable
 - expansion 25
- environment variable
 - GOBO_CC 1, 2, 3
 - GOBO_EIFFEL 3
 - GOBO_MT 2
 - set 53
- eposix.ecf 5
- epxc 10
- EPX_CGI vi, 79
- EPX_CURRENT_PROCESS 15, 31, 46
- EPX_DIRECTORY 102
- EPX_EXEC_PROCESS 15, 42
- EPX_FILE_DESCRIPTOR 14, 15
- EPX_FILE_SYSTEM 14, 15
- EPX_HTTP_10_CLIENT 60
- EPX_HTTP_CONNECTION 64
- EPX_HTTP_SERVER 61, 62, 64
- EPX_IRC_CLIENT 66
- EPX_LOG_HANDLER 76
- EPX_MIME_EMAIL 68
- EPX_MIME_PARSER 54
- EPX_MIME_PART 54, 68
- EPX_PIPE 15
- epxs 10
- EPX_SELECT 99
- EPX_SMTP_CLIENT 67, 68
- EPX_SYSTEM 53
- EPX_XHTML_WRITER 79
- errno 8
- error handling 86
- execl 97
- execle 97
- execlp 97
- execv 97
- execve 97
- execvp 97
- EX_ERROR1 89
- EX_HTTP_SERVLET2 63
- exit 97
- EXP_FTP_CLIENT 60
- EX_SMTP_MAIL 68
- f
 - fchmod 97
 - fchown 97
 - fclose 97
 - fcntl 97
 - fcntl.h 97, 99
 - fdatasync 6, 6, 97
 - fdopen 97
 - feof 97
 - ferror 97
 - fflush 97
 - fgetc 97, 98
 - fgetpos 97
 - fgets 97, 98
 - file
 - read entire 23
 - filename manipulation 36
 - fileno 97
 - file pointer 25
 - flockfile 97
 - flush 32
 - fopen 93, 97
 - fork 97
 - fpathconf 97
 - fprintf 97
 - fputc 97, 99
 - fputs 97, 99
 - fread 97
 - free 97
 - FreeBSD 6
 - freopen 97
 - fseek 98
 - fsetpos 98
 - fstat 98
 - fsync 6, 6, 97, 98
 - ftell 98
 - ftruncate 98
 - ftrylockfile 98
 - funlockfile 98
 - fwrite 98
- g
 - geant 1
 - getc 98
 - getchar 98
 - getcwd 98
 - getegid 98
 - getenv 98
 - geteuid 98
 - getgid 98

- getgrgid 98
- getgrnam 98
- getgroups 98
- getlogin 97, 98
- getpgrp 98
- getpid 11, 98
- getppid 98
- getpwnam 98
- getpwuid 98
- gets 98
- gettimeofday 98
- getuid 98
- gexace 4
- glibc 6
- gmtime 98
- Gobo 21, 34
- grp.h 97, 98
- h**
- htonl 98
- htons 98
- HTTP 10
- i**
- inet_ntoa 98
- ioctl 98
- isatty 98
- ISE Eiffel 2
- ISE Eiffel 6.2 5
- k**
- KI_CHARACTER_INPUT_STREAM 21, 34
- KI_CHARACTER_OUTPUT_STREAM 21
- kill 98
- l**
- LDAP 73
- LDIF 73
- libposix_ise_msc.lib 3
- libposix_ise_msc.lib 3
- libposix_se.a 3, 14
- libmteposix_ise.a 2
- libmteposix_ise_msc.lib 3
- library.xace 4
- LICENSE v
- license v
- link 98
- lio_listio 98
- little endian 18
- localeconv 98
- locale.h 98, 99
- localtime 98
- lock 28
- lseek 98
- m**
- make.exe 2
- makelib.exe 3
- malloc 98
- max_open_files 53
- memchr 98
- memcmp 98
- memcpy 98
- memmove 98
- memset 98
- MIME 10
- minicom 32
- MIT License v
- mkdir 98
- mkfifo 6, 14, 98
- mkstemp 98
- mktime 98
- mlock 99
- mlockall 98
- mmap 99
- modem 32
- mprotect 99
- mq-receive 99
- mq_close 99
- mq_getattr 99
- mq_notify 99
- mq_open 99
- mq_send 99
- mq_setattr 99
- mqueue.h 99
- mq_unlink 99
- msync 99
- multi-threaded programming 3
- munlock 99
- munlockall 99
- munmap 99
- n**
- nanosleep 99
- netdb.h 97
- netinet/in.h 98, 99
- non-blocking i/o 21, 34
- ntohl 99

ntohs 99

o

open 99, 103, 104

opendir 99

openlog 99

Open Source v

p

PAPI_UNISTD 8

pathconf 99

path name 25

pause 99

perror 99

pipe 99

poll v

POSIX_ASYNC_IO_REQUEST 35

POSIX_BASE 9

POSIX_BINARY_FILE 21

POSIX_BUFFER 17, 17, 18

POSIX_CONSTANTS 10

POSIX_CURRENT_PROCESS 48

POSIX_DAEMON 74, 74

POSIX_DIRECTORY 40, 41, 97, 99

POSIX_EXEC_PROCESS vi, 44

POSIX_FILE 21, 21

POSIX_FILE_DESCRIPTOR 15, 27,
92, 97

POSIX_FILE_SYSTEM 38

POSIX_FORK_ROOT 11, 48

POSIX_MEMORY_MAP 19, 99

POSIX_PERMISSIONS 39, 40

POSIX_SHARED_MEMORY 17

POSIX_SHELL_COMMAND 42

POSIX_SIGNAL 100

POSIX_SIGNAL_HANDLER 46, 47

POSIX_STAT 40

POSIX_STATUS 40, 98, 100

POSIX_SYSTEM 53, 100

POSIX_TEXT_FILE 21, 28

POSIX_TIMED_COMMAND 96

printf 99

p_stdio.c 94

p_stdio.h 94

putc 99

putchar 99

PUT request 62

puts 99

pwd.h 97, 98

r

raise 99

rand 99

read 99

readdir 99

realloc 99

redirect standard error 32

remove 99

rename 99

REST 62

rewind 99

rewinddir 99

rmdir 99

s

scanf 99

seek 25

select 99

semaphore.h 99

sem_close 99

sem_destroy 99

sem_getvalue 99

sem_init 99

sem_open 99

sem_post 99

sem_trywait 99

sem_unlink 99

sem_wait 99

servlet 62

setbuf 99

setgid 99

setjmp.h 101

setlocale 99

setpgid 99

setsid 100

setuid 100

setvbuf 100

shm_open 100

shm_unlink 100

sigaction 100

sigaddset 100

SIGCHLD 47

sigdelset 100

sigemptyset 100

sigfillset 100

sigismember 100

signal 100

signal.h 98, 99, 100

signal handler 46

sigpending 100

- sigprocmask 100
- sigqueue 100
- sigsuspend 100
- sigtimedwait 100
- sigwait 100, 103
- sigwaitinfo 100
- slash 25
- sleep 100
- SmallEiffel vi
- SmartEiffel 2
- Solaris 6
- sprintf 100
- srand 100
- src/library.xace 1, 6
- sscanf 100, 101
- stat 40, 100
- STC_TEMPORARY_FILE 13
- stdarg.h 101
- STDC_BASE 9
- STDC_BINARY_FILE 13, 30
- STDC_BUFFER 13, 17, 17, 91
- STDC_CONSTANTS 10, 13
- STDC_CURRENT_PROCESS 13
- STDC_ENV_VAR 13, 52
- STDC_FILE 21, 92, 97
- STDC_FILE_SYSTEM 13
- STDC_LOCALE_NUMERIC 98
- STDC_PATH 36
- STDC_SECURITY_ACCESSOR 91
- STDC_SHELL_COMMAND 13, 100
- stdc_signal_switch_switcher 5
- STDC_SYSTEM 13
- STDC_TEXT_FILE 13, 30, 31
- STDC_TIME 13, 97
- stderr 32
- stdin
 - binary 31
- stdio.h 94, 94, 96, 97, 98, 99, 100, 101
- stdio.h 98
- stdlib.h 96, 97, 98, 99, 100
- stdout 32
 - binary 31
- stream buffer 32
- strftime 100
- STRING 91
- string.h 98
- stropts.h 98
- support
 - commercial v
- SUS_BASE 9
- SUS_SYSLOG 76, 100
- SUS_SYSLOG_ACCESSOR 76
- SUS_TIME_VALUE 98
- sysconf 100
- syslog 100
- syslog.h 97, 99, 100
- sys/mman.h 98, 99, 100
- sys/select.h 99
- sys/socket.h 96, 97
- sys/stat.h 96, 97, 98, 100
- system 100
- system.se 14
- system.xace 4
- sys/time.h 98
- sys/utsname.h 100
- sys/wait.h 101
- t
 - tcdrain 100
 - tcflow 100
 - tcflush 100
 - tcgetattr 100
 - tcgetpgrp 100
 - tcsendbreak 100
 - tcsetattr 100
 - tcsetpgrp 100
 - temporary file 13, 84
 - terminal 29
 - password 29
 - termios.h 96
- TEST_IRC_CLIENT 67
- text mode 30
- time 100
- time.h 96, 97, 98, 99, 100
- timer_create 100
- times 100
- times.h 100
- tmpfile 100
- tmpnam 100
- ttyname 100
- tzset 100
- u
 - ULM_LOGGING 76
 - ULM_LOG_HANDLER 76
 - ULM_LOG_LEVELS 77
 - umask 100
 - uname 100
 - ungetc 100
 - unistd.h 96, 97, 98, 99, 100, 101

Unixmulti-threaded applications [1](#)unlink [8](#), [100](#)URI [10](#), [61](#)utime [100](#)utime.h [100](#)UT_URI [61](#)**v**vfprintf [100](#)Visual Eiffel [vi](#)vprintf [100](#)vsprintf [101](#)**w**wait [101](#)waitpid [101](#)Windows 2000 [6](#)WINDOWS_PAGING_FILE_SHARED_MEMORY
[19](#)WINDOWS_SYSTEM [53](#)write [101](#)**x**XM_UNICODE_CHARACTER_CLASSES
[5](#)

