

The un*x to OS/2-EMX Porting FAQ

Alexander Mai

(c) in 1999-2003 by Alexander Mai

Table of Contents

1 The un*x to OS/2-EMX Porting FAQ.....	1
1.1 Preface.....	1
1.1.1 What Is It?.....	1
1.1.2 How To Use This FAQ?.....	1
1.1.3 How To Report Bugs?.....	1
1.2 Copyright, License, Legal Stuff.....	2
1.3 News.....	2
1.3.1 General News.....	2
1.3.2 Latest Changes.....	2
1.4 The Golden Rules.....	3
1.5 Tools.....	3
1.5.1 Standard Tools.....	4
1.5.2 Scripting Languages.....	5
1.5.3 Archiver Formats.....	5
1.5.4 Helpful Stuff.....	6
1.6 Common APIs.....	7
1.6.1 Useful & Dirty Workarounds.....	7
1.6.2 APIs that need Special Care.....	8
1.6.3 Ported APIs.....	13
1.7 un*x Process Stuff.....	14
1.7.1 Starting Processes.....	15
1.7.2 Process Properties.....	16
1.8 un*x Filesystem Stuff.....	17
1.9 Miscellaneous Issues.....	19
1.9.1 Various Hints.....	19
1.9.2 sh vs. cmd.exe.....	22
1.10 In & Out.....	24
1.10.1 Device Driver Interfaces.....	24
1.10.2 Printing.....	24
1.10.3 Sound.....	24
1.10.4 Miscellaneous I/O.....	25
1.11 EMX & OS/2 Specifics.....	25
1.11.1 From EMX docs.....	25
1.11.2 From other sources.....	26
1.12 Configuration Stuff.....	27
1.12.1 Makefiles.....	27
1.12.2 Batch Files.....	28
1.12.3 auto* tools.....	28
1.12.4 Imakefile.....	30
1.13 X11 Specifics.....	30
1.14 External Resources.....	35
1.14.1 Related Documents.....	35
1.14.2 Online Resources.....	35

Table of Contents

2 Printing from non-PM apps.....	37
2.1 Direct Printing.....	37
2.2 LPR, LPD.....	37
2.3 Wrappers, Utils.....	38
3 Debugging.....	39
3.1 Tools.....	39
3.2 Procedure.....	39
3.3 Advanced Issues.....	40
3.3.1 Resolving X Errors.....	41
3.3.2 Well-known Tricks.....	41
3.4 Compile Time Problems.....	42
3.5 Portability Issues.....	43
3.5.1 Language Level.....	43
3.5.2 "Bitness".....	43
3.6 Bug Reporting.....	44
4 Debugging on OS/2.....	45
4.1 Tools and Helpers.....	45
4.2 Debugging Basics.....	45
4.3 Misc Hints.....	46
4.4 Other Resources.....	47
5 Regular Expressions.....	48
5.1 The Metacharacters of Regular Expressions.....	48
5.2 The Syntax of Regular Expressions.....	48
6 Ported Software Sites.....	49
6.1 Collections.....	49
6.2 Developers' sites.....	49
7 Standards & Docs.....	50
7.1 Official Standard Documents.....	50
7.1.1 C.....	50
7.1.2 C++.....	50
7.1.3 Fortran.....	50
7.1.4 un*x & related API References.....	50
7.1.5 Library References.....	51
7.1.6 Misc Programming.....	51
7.2 FAQs.....	51
7.3 Implementations.....	52
7.4 "Non-Programming" Standards.....	52
7.5 Definitions.....	54

1 The un*x to OS/2–EMX Porting FAQ

1.1 Preface

First I'm going to answer some questions that nobody ever asked ...

1.1.1 What Is It?

This is a *collection of tips & tricks* for *porting software from un*x* (here the term "un*x" means *UNIX* and UNIX-like systems") to IBM's *OS/2*. If you need a more thorough introduction into the business of porting, you should read the *draft* for the *UNIX/POSIX to emx–OS/2 Porting Guide* written by *Arnd Hanses*: it's also available in a local [PDF](#) version.

In general porting requires a broad range of knowledge, so this document also includes general information about coding techniques, un*x & X11 programming, and much more. I assume that the reader is already familiar with OS/2 and has some experience with un*x (usage at least, perhaps even system programming), C programming and EMX/gcc. Otherwise you can't do anything worthwhile here... *Sorry!*

1.1.2 How To Use This FAQ?

There is a [PDF version](#) of this document available . This is the *only* "official" version since it features a *Table of Contents* and is complete within a single file. You can even create printouts and extract a plain text version. (try `pdftops` and `pdftotext`, which are part of the [xpdf](#) distribution.) Remember that the *FIND* function of your browser/viewer is your friend! Some parts have been put in external documents with "local linkage", i.e. "compiled" ("official") versions already include those. Due to the substantial amount of external links within this FAQ access to the Internet is nearly mandatory.

1.1.3 How To Report Bugs?

In case you are going to correct an error, contribute another entry or fix spelling or grammar, please check first the primary location of this FAQ at

<http://www.lesstif.org/~amai/os2/html/porting.html>

for the latest revision. You may [send me an email](#) with corrections, enhancements and criticism!

1.2 Copyright, License, Legal Stuff

Contributions to this document came from various people and resources on the net. However to avoid that a *single one* is neglected in a reference or credits list, *all* are omitted! Two exceptions are being made for two individuals without their efforts I would not use OS/2 anymore and therefore not work on this FAQ: *Eberhard Mattes* (author of the EMX package including the docs which I frequently cite) and *Holger Veit* (who ported [XFree86 to OS/2](#)). Many hints given in this document came from these two people and the numerous members of their projects' [mailing lists](#). This document is being made available as **copyrighted freeware** (copyright holder being me, *Alexander Mai*): You may get this document and re-distribute it or parts of it as long as you don't charge people for the resulting work! If you re-distribute a substantial part of this FAQ you have to mention my copyright notice in a *credits section*!

Usage of the tips & procedures described here is totally on your own risk, there is no warranty that they do what you might expect without any unwanted side effects!

1.3 News

1.3.1 General News

Since I do not get much feedback concerning this document it is growing slowly, and also still contains lot of typos and errors. Currently (Q4/2003) the document is quite "stable", I'm busy fixing errors and broken links.

Work is/was ongoing to get many of the hints and workarounds mentioned here into a powerful **extension library for EMX** (AKA "libext"). Take a look and [see what's already done](#)! Especially try to look up whether symbols missing in this FAQ are already included there.

1.3.2 Latest Changes

[Last Update](#) (I may list some of the previous changes as well):

- Caught some grammar/spelling errors
- Updates to the standards document
- Update many external links
- Update some of my links and my email address (For checking links I'm using [ALC](#) Alexander's Link Checker)

1.4 The Golden Rules

There are some famous Golden Rules out there. The first one is quite simple:

Check out what's already available!

This is also known as:

Do not re-invent the wheel!

Many programmers waste time doing things over and over again, and especially programmers for OS/2 to enjoy this (just look at all those coding the very same kind of utilities over and over again, like file managers, thousands of wget clones, etc). Before you start coding search on [OS/2 software sites google](#) (or similar search engines) and asked on the EMX, XFree86OS/2 and related [mailinglists](#).

Enhance portability when porting!

Do not produce code which in turn does no longer build on the platform where it comes from. Minimum requirement is to use the

```
#ifdef __EMX__  
...  
#endif
```

clause (I'm talking about C/C++ only here). You should however even try to go a step further and use *generic catches* like:

```
#ifndef HAVE_NON_STANDARD_FEATURE_FOO  
...  
#endif
```

1.5 Tools

To actually port something you need to have *EMX* (see [OS/2 software sites](#)) and all important related development packages installed properly (in their latest version). To deal with X11 applications you need the [XFree86OS/2](#) package including the development stuff (all older commercial alternatives like IBM's PMX or Hummingbirds Exceed are abandoned/outdated and therefore neglected here. [Project Everblue](#), an X11 to PM mapper for local apps, is still in very early stage of development and of no use so far). At least for the X server itself you have more choices in addition to [XFree86OS/2](#):

The un*x to OS/2-EMX Porting FAQ

- [Hob X11](#)
- [WeirdX \(Java X Server\)](#)

Also check out [libext](#). In addition to providing libc enhancements it also ships with a set of utilities.

gcc 2.8.1, the default C compiler from the current EMX 0.9d distribution, is quite old. However meanwhile there are alternatives. I suggest to install *pgcc* which you may download from [OS/2 software sites](#) or [it's homesite](#). The *Pentium optimized GCC* offers OS/2 users access to a gcc of the 2.9x series, as opposed to EMX which is at gcc level 2.8.1. PGCC/2 also offers new features: improved code generation for recent processors, a new version of GAS (=GNU assembler) with MMX support, less C++ -related bugs, enhanced C++ features/language set and a new, but broken *g77* (=GNU Fortran 77 compiler).

Recently there also came up a [port \(beta status\)](#) of [gcc 3.x](#) (a mor recent version is available from [Innotek](#). Probably won't bring much for plain C programming, but certainly has enhanced support for C++.

There are various tutorials on getting EMX and XFree86OS/2 to work, please refer to the according [link section](#).

1.5.1 Standard Tools

Except the basic requirements as outlined above you will need additional tools if you want to build packages bigger than the famous "Hello_World.c" example. The following list is probably too long, but except a waste of disk space I see no reason not to install some utilities in advance ;-)

Those tools have to be first in %PATH% before any other [incompatible versions](#)!

All should be available from [OS/2 software sites](#).

- GNU file utils (cp, install, [ln](#), ls, mkdir, mv, rm)
- GNU shell utils (expr, pwd, sleep, tee, [test](#))
- GNU text utils (cat, head, join, sort, tail, uniq)
- find
- bash, ash (or other standard-like shell)
- GNU bison ([yacc](#) replacement)
- flex ([lex](#) replacement)
- grep (egrep, fgrep, etc.)
- [gzip](#) ("GNU zip")
- (GNU) Make
- [patch](#)
- [sed](#)
- [tar](#)
- uname (unfortunately many versions exist; get one that is compatible with your [autoconf](#) version)
- uudecode/uuencode

The un*x to OS/2-EMX Porting FAQ

Don't forget to put `sh.exe` into the `/bin` subdirectory on your working drive to keep many scripts/Makefiles running without changes (this holds for other standard tools as well, like `rm`, `cp`, ...).

Other useful tools include:

- `diff` (to create patches, so called "diffs", for textfiles only)
- `groff`, `man` (to read [un*x](#) man pages)
- `indent` (to format source code)

1.5.2 Scripting Languages

[un*x](#) offers a variety of script languages like `perl` and `Tcl/Tk`. Some are used for rather simple tasks if the more basic tools (like `sed`, `awk`, etc.) are no longer sufficient or the suitable command lines become too awful. The OS/2 command shell `cmd.exe` can not execute such things directly, so you either have to explicitly call the interpreter (like `foo bar.foo`) or rename it to `foo.cmd` and add a suitable first line containing `extproc foo` like e.g. for `perl` scripts:

```
extproc perl -S
```

Perl scripts of this kind have to be in your `%PATH%`!

The following table contains several language interpreters which have already been ported to OS/2. The offered links may not necessarily point to the latest version, but you will find out yourself.

Script languages and their OS/2 ports

Language	OS/2 port
<code>awk</code> (e.g. GNU <code>awk="gawk"</code>)	OS/2 software sites
<code>m4</code>	OS/2 software sites
<code>perl</code>	ftp.math.ohio-state.edu or numerous versions from OS/2 software sites
<code>python</code>	www.python.org and another page
(un*x) shell script	various ports; see sites.html
<code>Tcl</code>	www.vaeshiep.demon.nl
<code>Tk</code>	www.vaeshiep.demon.nl

1.5.3 Archiver Formats

Here is a list of some common archiver formats and the according tools to extract them (see [OS/2 software sites](#)).

The un*x to OS/2-EMX Porting FAQ

On un*x it's common to use cumulative suffixes, so `hello.tar.gz` refers to a file `hello.tar` which is an archive file produced by `tar` which has finally been packed by `gzip`.

Archiver formats

Extension	Related Tool
.bz2	bzip2
.cpio	<code>cpio</code>
.deb	Package for Debian linux
.gz	(GNU) gzip [<code>gzip -d</code> , <code>gunzip</code>]
.rpm	RPM ; OS/2 port of this tool
.shar	<code>sh</code> (shell archives) [<code>sh foo.shar</code>]
.tar	<code>tar</code>
.uue	<code>uudecode/uudeview</code>
.Z	<code>decompress/gunzip</code>
.z	<code>pack/unpack</code>
.zip	Info-(un)zip

1.5.4 Helpful Stuff

Here I present some tools which you don't really need, but which are nice to have around!

- There are numerous utilities available which translate files from un*x to DOS [text format](#) (and Macintosh if required) and vice versa: `dos2unix/unix2dos`, `recode`, `tr`, `trans`, ...
- TVFS: Toronto Virtual File System
Available from [OS/2 software sites](#). It installs a virtual drive and allows to create links on it. So you can access all your partitions on that single virtual drive and don't need to use drive letters!
- OS2TRACE (OS/2 API tracer; see [OS/2 software sites](#))
A tool which can trace down most OS/2 API calls.
Not very helpful when starting work on a port ...
- Get `chk4dlls` somewhere. It lists all DLLs required by an executable/DLL and is an equivalent to the `ldd` utility available on many un*x systems.
Note that it doesn't list libraries which are explicitly *dynamically* loaded during runtime using `DosLoadModule()`, `dlopen()`, etc.
- Get some tools to process the various documentation formats. Converters are available for
 - ◆ HTML to IPF
 - ◆ man to HTML
 - ◆ Texinfo to html ([new makeinfo](#))
 - ◆ [Texinfo to IPF](#)

Those converters may be combined in a tool chain, so that a conversion of some lengthy

The un*x to OS/2-EMX Porting FAQ

man page to a simple-structured INF document is actually possible.

- Get [CVS](#) (Concurrent Versions System): A tool used for many open source projects on the net.

You do not need to actively participate in the development to benefit from CVS: You can then easily get upgrades without downloading the whole package.

The given link may carry outdated executables; check out [OS/2 software sites](#).

- Get *ctags*.

It scans the sources and produces a reference list of all symbols together with their precise location. Together with a suitable editor (like [NEdit](#), [Emacs](#), [XEmacs](#), [vi](#) and it's clones, ...) this makes a powerful combination: e.g. you select a function's name and a single keystroke will bring you to its definition!

The probably best, because most powerful implementation is [Exuberant ctags](#).

1.6 Common APIs

Beyond the problems of getting configuration and Makefiles to run properly on OS/2 the most striking problem is obviously to get the missing un*x API calls done on OS/2. In general there are two solutions:

- map them to native interfaces
- forget about them

The first one sounds like the preferable approach, but on a single-user, non-un*x system it doesn't make much sense to deal in detail with concepts which don't exist on the target platform!

1.6.1 Useful & Dirty Workarounds

These are not necessarily "*canonical*" and clean approaches, but often they fulfill their purpose very well. This section includes functions, macros and constants. The functions/macro replacements might be "enhanced" by adding the correct number of arguments. Obviously defining a macro/function to either 0 or the appropriate return code upon error relies on the concept that the source was written properly and to some extent handles failures of these APIs properly.

Attention: using these redefinitions will make debugging more complicated since you have to remember all of them! At least I don't have a source code debugger which offers additional support here.

```
#define chdir                _chdir2           /* see below */
#define chown(x,y,z)        (0)
#define endgrent()          /* is type void */
#define FNDELAY              O_NONBLOCK
#define getcwd              _getcwd2         /* see below */
#define getdtablesize ()    (1024)
#define getgrent()          ((group*)NULL)
#define link(x, y)          (-1)           /* no hard links, below */
#define lstat(n, b)         stat(n, b)      /* see stat(), links */
```

The un*x to OS/2-EMX Porting FAQ

```
#define major(dev)          (0)          /* see below */
#define makedev(dev,minr)  (0)          /* see below */
#define minor(dev)         (0)          /* see below */
#define mkfifo(p,m)        (0)          /* see below for mkfifo() */
#define mknod(p,m,d)       (0)          /* see below */
#define mlock               (-1)        /* see below */
#define munlock             (-1)        /* see below */
#define munlockall         (-1)        /* see below */
#define readlink(x,y,z)    (0)          /* or (-1) and use errno? */
#define setgrent()          /* is of type void */
#define setpgrp()           (0)          /* ? */
#define sigset(x,y)         signal(x,y) /* well, almost ... */
#define strcasecmp          strcmp       /* see libExt */
#define strncasecmp         strncmp     /* see libExt */
#define symlink(x,y)        (-1)        /* see below */
#define sync()              /* check out fsync(), etc. */
#define S_ISLNK(x)          (0)          /* see below */
#define S_IFLNK             (0)          /* see below */
#define S_ISBLK(x)          (0)          /* see below */
#define S_IFBLK             (0)          /* see below */
#define S_ISVTX             (0)          /* see below */
#define usleep(t)           _sleep2(((t)+500)/1000) /* see below */
```

1.6.2 APIs that need Special Care

This section features some comments on available APIs which have to be handled with special care and some thoughts on not yet implemented ones.

ecvt(), *fcvt()*, *gcvt()*

Convert a floating-point number into a string. Not available as far as I know. Try replacing it by `*printf()` or even attempt to write your own version. Shouldn't be too hard ...

errno

`errno` is nothing un*x-specific, but an integral part of ANSI/ISO C's error handling. The catch here is that some programmers erroneously `#define errno` to something like `int` or even abuse it as a name for an identifier.

Related to `errno` is an important thumb-rule: always compile and link your EMX apps as multi-threaded applications (gcc flag `-Zmt`). This will avoid much trouble.

exec()*

See `fork()` and the section about [process](#) handling.

Replace `exec*()` with `spawn*()` and `exit()` if the parent process waits for the termination of the new process (by calling `wait()` or by waiting for `SIGCHLD`). This is required to keep the process ID of the child process. In a forked process, however, you don't have to do this because `emx.dll` does it for you.

fchmod()

Part of [libext](#).

fcntl()

`F_SETLK` is not supported.

Write your own `fcntl()` or add code which makes use of an existing `flock*()`

The un*x to OS/2-EMX Porting FAQ

implementation. Or watch [libext](#).

flock()

Check out the native API `DosSetFileLocks()`. Also see [libext](#).

fork()

See [exec*\(\)](#) and the section about [process](#) handling.

Replace `fork()` and [exec*\(\)](#) with [spawn*\(\)](#). Under OS/2, `fork()` is inefficient. You may not care if it's rarely being called, but read the docs for any specific information about the `fork()` implementation.

Don't forget that `fork()` doesn't work in `link386 (-Zomf)` created executables!

Read the section about [XTHREADS](#) if you are going to use `fork()` or threads within an X11 application.

getenv()

See section about [environment variables](#).

getrlimit()

This is not a [POSIX.1](#) function, but should be part of [libExt](#)

Also see `ulimit(3)`, `sysconf(3)`, *emxdev*: chapter "7 Customizing".

mlock(), *munlock()*, *munlockall()*

The `DevHlp*` interfaces (API for device drivers) provide such calls, to mark memory pages as non-swappable. However this is no trivial replacement which could be implemented by inserting a few lines of code.

poll()

A call similar to [select\(\)](#). Check out [libExt](#) for an emulation using this relationship.

popen()

Ensure to use the "b" in the mode flag when necessary!

Also see [system\(\)](#).

random()

The *EMX docs* claim that `random()` is superior to `rand()`, so try to use this instead. It's hidden in `libbsd`.

realpath()

[libext](#) has an implementation which can be used as a stand-alone version after editing a few lines.

rename()

EMX' `rename()` doesn't replace existing targets.

It also fails on open files.

Also see [remove\(\)](#).

remove()

`remove()` fails on open files.

On un*x you can (re)move open files. On OS/2 you can not and therefore this call might fail (this is a simplified picture; it can also fail on un*x – but won't do that often). A major component here is the file system, and since modern operating systems can deal with many of them, it is not precise to make statements based only on the OS being used.

select()

`select()` is a common source of portability problems. These may involve: the necessary header files, argument types or some "EMXisms", like the following: `select()` will report that `stdin` is "ready", i.e. will return immediately. If you want to

The un*x to OS/2-EMX Porting FAQ

trigger on any new input, i.e. keystrokes, you have to change the terminal settings (see [tcsetattr\(\)](#)). A small code snippet should give you the idea:

```
#include <sys/types.h>
#include <stdio.h>
#include <termio.h>
#include <termios.h>
#include <sys/time.h>
#include <sys/select.h>

int nfds, retval;
fd_set rfds;
struct timeval tv;
struct termios tio;

/* initialize rfds to 0: */
FD_ZERO(&rfds);
/* Watch stdin to see when it has input: */
FD_SET(fileno(stdin), &rfds);
/* only one file descriptor to monitor: */
nfds=1;
/* Now change EMX' default settings: */
tcgetattr(fileno(stdin), &tio);
tio.c_lflag &= ~IDEFAULT;
tcsetattr(fileno(stdin), TCSANOW, &tio);
/* Wait up to five seconds: */
tv.tv_sec = 5; tv.tv_usec = 0;
retval = select(nfds, &rfds, NULL, NULL, &tv);
if (retval > 0) {
    printf("Data is available now.\n");
    /* FD_ISSET(0, &rfds) is 'true' here now */
}
}
```

sigaction()

Check the code which signal model is used and choose the appropriate linker flags (see [signal\(\)](#)). Especially check if you need to link with `-Zbsd-signals` while [XFree86OS/2](#) ([imake](#); perhaps the docs mention it as well) still tends to use `-Zsysv-signals`.

The *EMX docs* describe what happens if `sigaction()` and `signal()` are used together.

signal()

By default EMX' signal processing is different from any existing standard (as of [SVID](#) or [BSD](#)): `SIG_ACK` should be used instead of the signal handler address to re-enable a signal by calling `signal()` when the signal handler has been called. This behavior can be changed with the `-Zbsd-signals` and `-Zsysv-signals` linker options of EMX gcc. If you use [POSIX.1](#) functions for signal handling, `SIG_ACK` is not required.

Comment out or replace code which is based on non-implemented signals. The EMX (IBM toolkit) docs list all available signals and give further information. But, of course, you need to know what you're going to miss then ...

By collecting signal types from various operating systems I got a rather long list – see below. The references {OS/2-IBM, [EMX](#), [Both](#), none} may be incomplete or even

The un*x to OS/2-EMX Porting FAQ

wrong. Please check the according docs, or even the headers!

Collection of signal types

SIGNAL name	Description	Reference	Comments
SIGABRT	Process abort signal	B	e.g. by <code>abort()</code>
SIGALRM	Alarm clock	E	
SIGBREAK	CTRL-BREAK by user	B	OS/2 only
SIGBUS	Bus error	E	
SIGCHLD (SIGCLD)	Child process stopped/terminated	E	
SIGCONT	Continue if stopped	-	
SIGDANGER	Danger signal	-	
SIGEMT	Emulator trap (results from certain unimplemented instructions)	E	
SIGFPE	Floating point exception	B	
SIGHUP	Hangup	E	
SIGILL	Illegal instruction	B	
SIGINFO	Information request	-	
SIGINT	Interrupt from terminal	B	
SIGIO	IO now possible	-	(4.2 BSD)
SIGIOT	IOT trap	-	
SIGGRANT	Monitor mode granted	-	
SIGKILL	Kill process	E	cannot be caught or ignored
SIGLOST	Resource lost	-	
SIGLWP	Used by thread library.	-	
SIGMSG	Monitor mode data available	-	
SIGNOFP	Floating point co-processor not available	-	
SIGPHONE	Line status changed	-	
SIGPIPE	Broken pipe	E	write to pipe without reading process
SIGPOLL	I/O possible	-	
SIGPROF	Profiling timer expired	-	
SIGPWR	Power failure	-	

The un*x to OS/2-EMX Porting FAQ

SIGQUIT	Quit from terminal	E	
SIGRETRACT	Need to relinquish monitor mode	-	
SIGSEGV	Segmentation fault	B	
SIGSOUND	Sound completed	-	
SIGSTKFLT	Stack fault on coprocessor	-	(linux)
SIGSTOP	Stop process	-	
SIGSYS	Illegal system call	E	
SIGTERM	Termination	B	
SIGTRAP	Breakpoint/tracepoint trap	E	(see hard-coded breakpoint)
SIGTSTP	Stop typed at tty	-	
SIGTTIN	tty input for background process	-	
SIGTTOU	tty output for background process	-	
SIGUNUSED	Unused signal	-	a very important one ;-)
SIGURG	Urgent I/O condition	-	
SIGUSR1	User defined signal #1	B	
SIGUSR2	User defined signal #2	B	
SIGUSR3	User defined signal #3	I	EMX seems to miss this!?
SIGVTALRM	Virtual timer expired	-	
SIGWINCH	Window resize	E	see termsize lib
SIGWIND	Window change	-	
SIGXCPU	CPU time limit exceeded	-	
SIGXFSZ	File size limit exceeded	-	

`stat()`, `fstat()`, `lstat()`

A call to query about an entry in your filesystem. This may be non-portable when it checks for non-existing properties on the new platform, like special file types or even low-level entries within the file system internals (inodes, etc.).

`lstat()` may be mapped to `stat()`, but for compatibility with future enhancements it should have its own wrapper (see [libext](#)).

`system()`

If you are lazy and keep the un*x-like forward slashes in paths arguments for `system()` calls may be passed to `cmd.exe` which might not be able to handle them!

`system()` and `popen()` do not expand wildcards unless `COMSPEC/EMXSHELL` points to a shell which expands wildcards (of course on un*x those calls wouldn't do either – but the shell being called). Read the *EMX docs* to learn which shell is called by these routines! One may run across code like `system("foo&");` which would try to launch

The un*x to OS/2-EMX Porting FAQ

a job in the background with an un*x shell but this won't work with `cmd.exe`. See "[sh vs. cmd.exe](#)"!

`tcsetattr()`

Any kind of subtle problems with VIO programs may stem from the fact the EMX by default uses a non-standard mode for terminal IO. The first call to `tcsetattr()` switches to Posix mode.

`unlink()`

An obsolete call. Use `remove()` instead.

`usleep()`

The workaround given above (using `_sleep2()`) doesn't feature a microsecond resolution; but contrary to its name many implementations on un*x won't do either. More info on timers on OS/2 is available on [EDM/2](#). Also see info about [g/setitimer\(\)](#).

Another quite portable approach uses `select()` as shown in the example code below. You must not specify descriptors to monitor, but only a timeout. Check out the docs which resolution actually is provided here and about properties of this call in general (e.g. the effect of signals being raised).

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/select.h>

struct timeval tv;
long secs, usecs;

/* set the interval: seconds and microseconds */
tv.tv_sec = secs;
tv.tv_usec = usecs;

/* tell select() to monitor no file descriptors */
select(0, (fd_set *)NULL, (fd_set *)NULL, (fd_set *)NULL, &tv);
```

1.6.3 Ported APIs

This is a collection of known ports of APIs which are more complex than a simple wrapper like:

```
#define foo bar
```

I try to list only "general purpose" stuff that can be used in any kind of application. Interfaces for rather special purposes may be collected elsewhere. Some of those and many others are meanwhile part of [libExt](#).

- `dlopen()/dlopen()` (access to dynamic link libraries)
Part of [libExt](#).
- `drand48()`, `srand48()` (random numbers)
Part of [libExt](#).
You can try to write a simple wrapper around `rand()` or even better `random()`. Just be aware of the slightly different interface.
- `gettext()` (National Language Support library from GNU)

The un*x to OS/2-EMX Porting FAQ

(on [OS/2 software sites](#) and from H.C. Chu)

- `mmap()`, `munmap()`, etc. (mapping files into memory)

A work in progress implementation is part of [libExt](#) and a closed-source one on [OS/2 software sites](#).

There's an [article](#) explaining how to write your own implementation.

- `mkfifo(3)` (a named-pipe mechanism using the file system)

An implementation comes with the math package `Octave` (on [OS/2 software sites](#)).

- `pthread` (POSIX threads) (on [OS/2 software sites](#) and on [ftp.everblue.org](#))
- (GNU) `readline` (provides simple editing features in an input line on terminals)

Implementations are on [OS/2 software sites](#), but are not usable in `xterms`.

- `regex` ([regular expressions](#)) (on [OS/2 software sites](#); also part of [libext](#))

A *Regular Expression* is a text string that describes some set of strings. You can use it to search (and afterwards edit/process) for certain patterns. See [regex.html](#) for a small summary.

Be aware that beyond some [common features](#) there are portability problems waiting if you use it heavily.

- `rpc` ([Remote Procedure Call](#)) interfaces are supplied with [libExt](#).

H. Veit has done [a port](#) for EMX.

- `scandir()` (read directory entries)

Part of [libExt](#).

- `setitimer()/getitimer()` (timer functions)

- ◆ H. Veit posted some [comments](#) and an [example](#) how to implement it using a secondary thread.

- ◆ An (incomplete) implementation is part of [libExt](#).

- ◆ `hrtimer.sys` (version ≥ 1.1) with new simple `open()/read()` interface should make timing easy and enable writing an `itimer` compatible interface. Also you may read this [EDM/2 article](#).

- `setpriority()`, `getpriority()` (set/get process priority)

A simple mapping to OS/2 calls should be nearly straightforward, as this [sample](#) shows. Also available within [libext](#).

- `shm` (shared memory between different processes): `shmat()`, `shmctl()`, `shmdt()`, `shmget()`

Implementations come with [XFree86 OS/2](#) and [libext](#) as well.

- `syslog()` (send messages to system logger)

Various ports are available on [OS/2 software sites](#).

1.7 un*x Process Stuff

Process handling is a bit different on [un*x](#). However often you won't notice that, since the most frequently used APIs are dealing with starting a process via `system()`, `exec*()`, etc. which are available on OS/2 EMX (also see `fork()`).

The term *thread* has no unique meaning in the [un*x](#) world as opposed to OS/2. Those [un*x](#) systems which had already offered something similar to the OS/2 threads concept often developed their own standard for this (see [threads-FAQ](#) for more details). Finally `pthread` (POSIX threads)

The un*x to OS/2-EMX Porting FAQ

settled down as a standard. A wrapper library from this API set to native OS/2 threads exists (see [OS/2 software sites](#)), but may still be work in progress.

1.7.1 Starting Processes

For purposes similar to those of multi-threading often child processes started via `fork()` are being used.

When `fork()/exec*` is used to start a subprocess often communication between the parent and child process is done. The most simple way is to redirect `std*` (=stdin/stdout/stderr) using pipes (created by `pipe()`). If you migrate the code from `fork()/exec*` to `spawn*` you have to change this code slightly as well: in the former case pipes are created, then the child process switches it's `std*` to the according ends of the previously created pipes before calling `exec*`. The parent just closes the unused ends of the pipes and keeps those required to communicate with the child. When using `spawn*` you don't have explicit access to the child, but it inherits the pipes/handles given from it's parent. So you have to redirect `std*` of the whole process to the pipes, start the child and fixup `std*` again:

1. create backups of the `std*` handles (\rightarrow `dup()`)
2. redirect the `std*` handles (\rightarrow `dup2()`)
3. start the child (\rightarrow `spawn*`)
4. restore `std*` (\rightarrow `dup2()`) and `close()` unused file handles

An example (no warranty, but it seems to work) with those two alternatives is now given: it will execute the command given as the first argument. First the version using `fork()/exec*`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

    char data[1024];
    int fd[2], result;

    if (pipe(fd)) exit(1);           /* error creating pipe */
    fflush(stdout);                 /* flush stream before redirection */
    if ((result=fork())) {          /* parent process */
        if(result==-1) exit(2);     /* error fork()ing */
        else {
            close(fd[1]);           /* close writing end */
            setvbuf(stdout, NULL, _IONBF, 0); /* disable buffering */
            while ( read(fd[0], data, 1) ) {
                putchar((char)data[0]);
            }
            close(fd[0]);
        } /* parent */
    }
}
```

The un*x to OS/2-EMX Porting FAQ

```
else { /* child */
    close(fd[0]); /* close reading end */
    dup2(fd[1],1); /* connect writing end to stdout */
    close(fd[1]);
    execv(argv[1], argv+1);
    exit(3); /* exec*() returns on error only */
}
exit(0);
}
```

or with `spawn*()`:

```
#include <stdlib.h>
#include <stdio.h>
#include <process.h>
#include <fcntl.h>
#include <io.h>

int main(int argc, char *argv[]){
    char data[1024];
    int fd[2], ftmp;

    if (pipe(fd)) exit(1); /* error creating pipe */
    fflush(stdout); /* flush stream before redirection */
    ftmp=dup(1); /* save the stdout handle */
    dup2(fd[1],1); /* assign write end of pipe to stdout */
    close(fd[1]); /* close pipe handle*/
    setvbuf(stdout, NULL, _IONBF, 0); /* disable buffering */
    /* prevent unused pipe handles from being inherited by childs */
    fcntl(ftmp, F_SETFD, FD_CLOEXEC);
    fcntl(fd[0], F_SETFD, FD_CLOEXEC);
    spawnv(P_NOWAIT, argv[1], argv+1); /* start child process */

    /* restore the original stdout file handle and close pipe */
    dup2(ftmp,1);
    close(ftmp);
    /* demonstrate that things work as expected: read and print out something */
    read(fd[0], data, sizeof(data));
    printf("%s\n", data);
    close(fd[0]);
    exit(0);
}
```

Watch out for open files (socket, ...), better: file handles, being passed to subprocesses: e.g. handles to an open file which are being passed to a spawned child process may prevent the parent from removing that file later on. See `fcloseall()` and `fcntl()` (-> `FD_CLOEXEC` flag).

1.7.2 Process Properties

Nothing sophisticated from the un*x point of view is taught here, just briefly mention some interesting features from an OS/2 users point of view ...

The un*x to OS/2-EMX Porting FAQ

- Information about the processes on the system may be retrieved using the non-portable `/proc` interface.
- You may halt a process by sending `SIGSTOP` and make it continue by sending `SIGCONT`.

1.8 un*x Filesystem Stuff

Common "native" un*x filesystems are quite different from those usually found on OS/2 (FAT, HPFS, HPFS386, JFS). Be aware that some really surprising things may show up: interfaces to memory, processes and hardware devices for example!

Let's have a look at some interesting properties:

- File names are *case-sensitive*!
Working on `libExt` showed me that one has to do some search on native OS/2 APIs which can be used for case-sensitive file system stuff. `DosQuery*Info()` will succeed on calls for files in a case-insensitive matching! More useful is `DosFindFirst()` and it's friends (see Toolkit docs for sample code).
- *No drive letters* are being used!
- There are standard paths which always exist! Try out `man hier` on your favorite un*x system to get a description of the *file system hierarchy* being used. In the following overview I may also mention some which are not mandatory, while trying to have all the canonical ones included.

<code>/</code>	root directory. That one is unique due the absence of drive letters.
<code>/bin</code>	contains system executables
<code>/dev</code>	virtual interface to devices of all kinds (hardware, logical, ...)
<code>/etc</code>	configuration data
<code>/home</code>	directories of each user
<code>/opt</code>	optional installed stuff
<code>/proc</code>	Interface to processes on the system. By read and even write access to this virtual file system you have control over all processes. However there's no unique standard for this.
<code>/sbin</code>	system executables (administration stuff, ...)
<code>/tmp</code>	a place for temporary data
<code>/usr</code>	binaries, data for the users

The un*x to OS/2-EMX Porting FAQ

/var

partition which contains stuff of varying size during run time (logs, mails, data caches, ...)

- Distributed over a few places on the system you will find the *man* pages, the standard un*x online help format (a few GNU projects have chosen [Texinfo](#) instead nowadays: you shouldn't support this). Several subdirectories contain manuals which usually carry the section number as a suffix (like `foo.2`)

Section 1

user commands

Section 2

system calls

Section 3

libc calls

Section 4

devices (e.g. `hd`, `sd`)

Section 5

file formats and protocols (e.g. `wtmp`, `/etc/passwd`, `nfs`)

Section 6

games

Section 7

conventions, macro packages, etc.

Section 8

system administration

Section n

Tcl/Tk, Perl, etc.

- un*x file systems support *links*, e.g. you can have multiple references to a single file or directory (see [TVFS](#)). They can be created by the The un*x tool `ln`, which is therefore sometimes required by scripts. *Symbolic link* (also called *soft link*) are created by `ln -s foo bar`. *Hard links* are fortunately rarely used in applications and are very hard to fake.

Using a replacement for symbolic links which does actually copy and also echoes a warning might be a good, but "dangerous" idea. This approach will not only miss updates on that file but might also cause a program to produce unwanted files: applications sometimes create links for temporary file access. Performing a file duplication will create trash on your harddisk which is not being cleaned up.

- The separator of path components is a slash '/' as opposed to OS/2's backslash '\'. Many OS/2 APIs understand the '/', but `cmd.exe` and most apps do not.
- Multiple paths (e.g. in environment variables like [PATH](#)) are separated by a colon ':'.
 - The root dir on an un*x system is "/".
 - Often code checks whether a given path is an absolute one by comparing the beginning with "/". So *grep* for this string! (see `_fullpath()` and `_abspath()`)
- Using

```
#define getcwd _getcwd2
#define chdir _chdir2
```

The un*x to OS/2-EMX Porting FAQ

may help to support drive letters.

- Note that `"/abc"` is a valid un*x filename. It's equivalent to `"/abc"`. In general multiple `'/'` characters won't hurt, unlike the situation on OS/2.
- `chdir (".")` is a no-op under un*x if the current working directory is the root directory. Under EMX it fails in the root directory.

1.9 Miscellaneous Issues

This is a collection of various helpful suggestions which don't fit yet in the other sections.

1.9.1 Various Hints

- Static libraries on un*x follow the unique naming convention to carry a leading `lib` prefix, like `libfoo.a`. If you want to link against it however, the right command is `-lfoo`. On EMX unfortunately this prefix has been dropped, i.e. `-lfoo` links against `foo.a` (or `foo.lib`) instead.
- Shared and dynamic link libraries use the suffix `.so` on most un*x systems (alternatives include `.sl`). There's no need for an import library usually. On un*x/ELF systems shared/dynamic libraries may contain unresolved references, on OS/2 they at least carry a 'link' to the DLL which contains those. This has both advantages and disadvantages, e.g. if you want to access symbols from a dynamically opened library (using `dlopen()`): on OS/2 you will be able to call such a routine in any case, on un*x/ELF you might end up with unresolved symbols!
- On un*x you can start executables having any name – given the executable file bit is set properly. On OS/2 things are a bit more complicated. You're on the safe side if you adopt the convention used by `cmd.exe`, i.e. you can only execute files with the endings `.com`, `.exe`, `.cmd`, `.bat` (this is the exact search order, BTW!). Most of this trouble is bound to `cmd.exe` and all APIs which may (or may not, see docs) interface with it (including `popen()` and `system()`).
- For "fast" moving targets like `libpng` (image format library) or nameclash examples like `Xaw`, `Xaw3d` and `Xpm` don't be shy to link statically ...
- Check for those "un*xisms" (listed in this FAQ) by using `grep` or an equivalent command to do a recursive search. Look for all standard paths (see above), and all commands which are mentioned in this introduction.
- Famous problems are related to the binary/text mode mess on OS/2, especially when using `stdin/stdout`. See `_fsetmode()`, `fopen()`, ... in the *EMX docs*.
- Note that on un*x plain text files have a different format than on DOS, OS/2. There are many tools available to convert between them (see [tools section](#)). This also requires much care on the `*open()` calls.

The un*x to OS/2-EMX Porting FAQ

A short list of text formats

Operating systems	Line Delimiters (hex)	
DOS, OS/2	0D	0A
un*x		0A
Macintosh	0D	

- There are some environment variables which are frequently used on un*x, and often also on OS/2. Some un*x apps erroneously assume some of them to always exist, i.e. `getenv()` calls to return a non-NULL result. Changing this is a valuable bugfix, not a porting related change!

EDITOR

Your favorite editor.

BTW, in case you got trapped inside vi [a famous un*x editor], feel uncomfortable, but don't know how to get out, here's the magic code:

:q!

HOME

Directory which contains data specific for the current user

HOSTTYPE

Name of platform

LD_LIBRARY_PATH

On a couple of systems used to tell the loader where to look for shared libraries.

Equivalent on OS/2 would be `BEGINLIBPATH`

LOGNAME

Same as `USER` (??)

MACHTYPE

Description of architecture/OS

MAIL

Location of your incoming mailbox

OSTYPE

Type of operating system

PAGER

Standard tool to page output. Something like "more" or "less"

PATH

Same as on OS/2. Except the [delimiter character](#)

PRINTER

Default printer. Might be used in conjunction with printing tools like "lp", "lpr"

PWD

Current directory

SHELL

Standard shell to be used. Often used by Makefiles.

TERM

The un*x to OS/2-EMX Porting FAQ

Name of the terminal type currently used (see [termcap](#), [terminfo](#)). Make sure that your software runs in an OS/2 window as well as in an xterm!

TERMCAP

Used by termcap to find termcap.dat which describes the capabilities of the used terminal device.

Nowadays often a different system called "terminfo" is being used for the very same purpose.

TMP

Usually not necessary, since all systems have a directory for temporary stuff in /tmp. On OS/2 unfortunately one sometimes runs across TEMP, TMPDIR and other garbage.

TMPDIR

The environment variable which [POSIX](#) apps should use to determine the temporary directory.

USER

Name of user

WINDOWID

Id of current X11 window

You may use the existence of this variable to distinguish between an xterm and an OS/2 textmode session (OS/2 window or fullscreen).

- Usually `argv[0]` within `main(int argc, char *argv[])` contains the name of the current executable. This is sometimes checked and used to determine the runtime behavior (e.g. *egrep* and *grep* may be the very same binary or *xfoo* may be the X11 GUI version of some program *foo*). This check will always be against "foo", so you have to add a check for `foo.exe` as well.
- A famous [FAQ of un*x programming](#) is how to determine the name and absolute path of the current executable. While there is no simple approach, most methods include the usage of `argv[0]` (e.g. `basename(argv[0])`). On OS/2 the answer is simple: EMX provides the interface `_execname()` which is more or less just:

```
#include <stdlib.h>
#include <os2.h>

int _execname (char *dst, size_t size) {
    PTIB ptib;
    PPIB ppib;

    if (DosGetInfoBlocks (&ptib, &ppib) != 0) return -1;
    return DosQueryModuleName (ppib->pib_hmte, size, dst);
}
```

- Never ever name an executable `test { .com, .exe, .cmd, .bat }`!
`test` is a standard program on un*x, which does apparently nothing – at least if you don't have a closer look at it ... (yes, if you want to know more about the ellipses you should ask "man test" :-)
- Unfortunately some OS/2 tools (including built-in `cmd.exe` commands) have their counterparts on un*x with exactly the same name but different semantics/syntax.
Below's an incomplete(!) list. The "external" and "internal" is meant with respect to

The un*x to OS/2-EMX Porting FAQ

cmd.exe.

External commands

find, install, join (DOS), more, patch, sort, time, unpack

Internal commands

cd, date, dir, echo, exit, mkdir, rmdir, set

- To merge a.out objects you may try something like

```
ld -X -r -o darin.o foo1.o foo2.o ...
```

 However you will rarely need to do so. Better create static libraries/archives which collect object code in a much more convenient way.
- You will run across many new file formats. Get the [file](#) utility for OS/2 which can identify file types using a database of file signatures (mostly some unique header which is a few bytes long). This is a standard utility on un*x.
 Among the most important formats are [archiver](#) types, which are covered elsewhere in this FAQ.
- For better conformance to the [IEEE 754](#) standard you may use:

```
#include <float.h>
unsigned All_Exceptions =
    EM_INVALID|EM_DENORMAL|EM_ZERODIVIDE|EM_OVERFLOW|EM_UNDERFLOW|EM_INEXACT;
_control87(PC_53, MCW_PC);          /* IEEE compatible precision */
_control87(All_Exceptions, MCW_EM); /* don't mask, i.e. hide exceptions */
```

- /dev/random, a device which should deliver (pseudo) random numbers has no equivalent in the poor random number generators which are distributed with the various libc's on OS/2. For more info on random numbers visit the [/dev/random Support Page](#) and [The pLab Project Home Page](#). And the local information about the [random\(\)](#) and [*rand48\(\)](#) interfaces.

1.9.2 sh vs. cmd.exe

While many want to abandon the "good old", but brain-dead cmd.exe there are many reasons to keep it, mainly for compatibility to existing solutions. So one may have to migrate a couple of shell commands and even complete simple scripts.

- Remember the cmd.exe/sh operators (which are surprisingly often identical ...):

Meaning	Syntax	
	sh	cmd.exe
execute two commands <code>foo</code> and <code>bar</code>	<code>foo ; bar</code>	<code>foo & bar</code>
execute <code>foo</code> ; if successful, i.e. return code = 0 then execute <code>bar</code> as well	<code>foo && bar</code>	<code>foo && bar</code>
execute <code>foo</code> ; if unsuccessful, i.e. return code $\neq 0$, execute <code>bar</code> as well	<code>foo bar</code>	<code>foo bar</code>

The un*x to OS/2-EMX Porting FAQ

execute <i>foo</i> and pipe its stdout to stdin of command <i>bar</i>	<code>foo bar</code>	<code>foo bar</code>
run <i>foo</i> in the background	<code>foo &</code>	<code>detach foo</code>
redirect stdout from <i>foo</i> into file <i>bar</i>	<code>foo >bar</code>	<code>foo >bar</code>
redirect stderr from <i>foo</i> into file <i>bar</i>	<code>foo 2>bar</code>	<code>foo 2>bar</code>
redirect both stdout and stderr from <i>foo</i> into file <i>bar</i>	<code>foo 1>bar 2>&1</code>	<code>foo 1>bar 2>&1</code>
arguments inside shell script	<code>\$1 \$2 ...</code>	<code>%1 %2 ...</code>
environment variables	<code>\$foo</code>	<code>%foo%</code>
group commands together as a single one	<code>(ls ; cd ..)</code>	<code>(ls & cd ..)</code>
loop over some list	<code>for i in \$PATH_LIST; do (cd \$i && ls) ; done</code>	<code>for %i in (%PATH_LIST%) do (cd %i && dir cd ..)</code>

- All standard un*x shells expand wildcard characters:

*

match every set of characters

?

match a single character.

To get this done on OS/2 you have to do it "manually" in your code since `cmd.exe` won't do it for you. However be aware that even people on OS/2 may use shells which already expand (4os2? or an un*xish shell). This shouldn't hurt at first glance, but may end up with a substantial number of commandline arguments passed to your application! EMX (and even IBM's toolsets) has it's own routine to help you getting that job done:

```
#include <stdlib.h>
```

```
int main (int argc, char *argv[]) {
    _wildcard (&argc, &argv);
    /* ... the program ... */
}
```

- To avoid expansion of wildcard characters one may quote command line arguments within a "" pair. `cmd.exe` has a slightly different concept here. Anyway for both trivial examples include specifying file names with embedded blanks, e.g.

```
foo "file name with blanks.txt"
```

- `cmd.exe` is unable to accept environment variables with include some special characters like '='. So trying to specify

```
set EQUATION=E=mc2
```

does not work. A work-around is to use a REXX wrapper:

The un*x to OS/2-EMX Porting FAQ

```
/* Ok, use REXX */  
call Value "EQUATION", "E=MC2", "OS2ENVIRONMENT"
```

- Sometimes long command pipes or those which have to transfer large amounts of data fail on `cmd.exe`, e.g.

```
gzip -d foo.tar.gz | tar xf - | sort | bzip2 -dc >bar.bz2
```

may produce an error message like

```
SYS0008: There is not enough memory available to process  
this command.
```

Then try grouping commands together as [shown above](#).

1.10 In & Out

This is a section about general Input/Output related topics. Since that issues are rather complex I can't even nearly fill/discuss all subsections ...

1.10.1 Device Driver Interfaces

Under un*x device drivers frequently feature interfaces which show up in the file system. This is at first glance something new to OS/2 users, but actually they already know similar interfaces (`\PIPE\`). Since this mechanism to access arbitrary devices has no trivial mapping to OS/2 capabilities, not all [related interfaces](#) can be ported: the code needs a major rewrite.

1.10.2 Printing

Printing on un*x is quite different to OS/2: no sophisticated system interfaces exist (read: may exist but are rarely used and not necessarily portable) so all applications have to do this on their own. Existing no-cost solutions (since only those are – if at all – of interest to OS/2 users in this context) include:

- the "new" X11 printing interface (→ `libXp`, which in turn may be used by libs such as [M*tif](#)). This is not supported by [XFree86OS/2 3.x](#), though.
- clever tools which trigger a conversion of "arbitrary" input data to Postscript, like [a2ps](#).

As a common denominator usually [Postscript](#)-capable printer is supposed to be available to the system. Also a set of standard utilities to submit/manipulate jobs for this device is required.

If you're lucky enough to have a Postscript printer on your OS/2 system you may use the `lpr` and `lpd` tools (shipped with Warp 4) to access the Postscript printer queues from the commandline. More on this topic can be found in the separate document [printing.html](#) which however covers "only" printing from non-PM applications.

1.10.3 Sound

Portable audio doesn't seem to be available across operating system borders. For many un*x flavors there's the [Open Sound System](#) (OSS), which brings digital audio (also MIDI) with an "open" interface.

The un*x to OS/2-EMX Porting FAQ

There's a `/dev/audio` emulation available from [OS/2 software sites](#) (`devaudio*.zip`). Given this version one might imagine to have the current implementation enhanced by additional features.

1.10.4 Miscellaneous I/O

With respect to general hardware access it's hard to find a portable standard at all here. Here's a short list of docs/projects which deal with portable I/O:

- [serial IO](#)

1.11 EMX & OS/2 Specifics

Here I address peculiarities and problems that the EMX runtime and OS/2 system present to the programmer. Most are taken from "*emxlib.doc: 5.1 Porting Unix applications*" and from *emxdev.doc: 5 Hints for porting Unix programs to emx*", but I'm also adding entries here myself.

1.11.1 From EMX docs

Though the following entries are more or less excerpts from the *EMX docs* I've started to modify them and therefore do no longer claim that it's actually the same what you can read in the EMX distributions!

- Socket handles are not inherited across `exec*()` and `spawn*()`. A process created by `fork()` inherits the socket handles of its parent process, though.
- The size of messages is restricted to 32767 bytes (this is a limitation of IBM TCP/IP for OS/2).
- By default sockets are in binary mode. Use `setmode()` to switch to text mode. Text mode applies to `read()` and `write()`, only. `recv()` and `send()` always use binary mode.
- The functions `recvmsg()`, `sendmsg()`, and `socketpair()` are not implemented.
- Change all `open()`, `fopen()`, `fdopen()` and `freopen()` calls to use `O_BINARY` or `"b"`, respectively, for binary files (see "[text formats](#)"). If a file contains both binary and textual data, read the file in binary mode and do the conversion yourself. Usage of the `-Zbin-files` (gcc) linker flag is in general a bad idea since it too often breaks handling of text files.
- Though `fseek()` and `ftell()` now work on text files, the offsets are different from what Unix programs expect. You may have to open the files in binary mode and ignore carriage returns (this has been done in GDB).
- Programs reading `a.out` files should be changed to call `_seek_hdr()` or `_fseek_hdr()` before reading the header to support `.exe` files. More changes are usually required.
- The null device is called `/dev/null` under Unix. The `__open()` system call translates the filenames `/dev/null` and `/dev/tty` (lower case, with slashes) to `nul` and `con`,

The un*x to OS/2-EMX Porting FAQ

respectively. However,

```
system ("whatever >/dev/null");
```

won't work as the standard OS/2 interpreter (`cmd.exe`) doesn't recognize `/dev/null` (see [system\(\)](#)).

- Do not use the `PTRACE_TRACEME` request of `ptrace()`: use `P_DEBUG` instead when starting the process with `spawn*()`.
- The shell isn't called `/bin/sh`. Use [system\(\)](#).
- Printing single characters is inefficient. A solution is to use

```
setvbuf (stdout, NULL, _IOLBF, BUFSIZ);
```

and to use `fflush(stdout)`; if you need the output immediately (flushing is required only after displaying prompting texts before reading input or displaying progress reports that don't end with a newline character). GDB output has been made much faster by using line buffering.

- Note that `VEOF != VMIN` and `VEOL != VTIME`. Programs which use `VEOF` and `VEOL` to access `VMIN` and `VTIME`, respectively, should be changed to use `VMIN` and `VTIME`. `emx` uses separate fields for `VEOF`, `VEOL`, `VMIN` and `VTIME`.
- To use `termio`, you have to reset the `IDEFAULT` bit of `c_lflag`. This does not apply to `termios`.

Always use the standard interface, i.e. [POSIX.1](#) `termios` in new code!

1.11.2 From other sources

- When using `spawn*()` take care of unwanted effects when [file handles](#) are inherited. Also check out the related problem of [socket handles](#)!
- Use `termio` or `termios` or read the keyboard with `_read_kbd()` if you don't want to get input line by line.
Do not use `_read_kbd()`! This breaks your code in [xterms](#) and in addition it is non-portable!
- To figure out which preprocessor `#defines` are in effect you can use the example batch script (for `cmd.exe`) below. Two [un*x](#) utils (`touch`, `rm`) are required, but you may easily replace them by native OS/2 calls. Advantage of the given version is obviously that you can use it almost literally on many [un*x](#) system and it can handle a big number of additional commandline arguments like compiler flags: `"-ansi"` is a candidate here.

```
touch __shd.c
gcc -dM -E %1 %2 %3 %4 %5 %6 %7 %8 %9 __shd.c
rm __shd.c
```

- Sometimes projects link explicitly against `libc (-lc)`. This is 95% useless and sometimes will trigger problems even on [un*x](#). On EMX the situation is even worse, the required C runtime is distributed in more than a single library, consult the *EMX docs* on that issue.

The un*x to OS/2-EMX Porting FAQ

- The [un*x](#) math library (`libm`) is often specified in Makefiles among the linker flags. Using EMX you don't need this (the functions are already among the standard C libs). But to keep the linker happy with those Makefiles a dummy math library exists, which is linked if `-lm` is used. The enhanced math library from [libext](#) is available in a stand-alone library.
- `ranlib` (a tool to prepare static libraries for usage by linker) is not available and not necessary with EMX anyway. You may use the `s` command of `ar` instead, like `ar s foo.a` (or just forget about it completely ...).
- A famous issue on OS/2 programming is the task to allocate more than 32MB of memory for a single object. Yes, of course this is possible, but when using EMX it may not be straight-forward. Check out the `_uflags()` call. Note that it interferes with `fork()`.

1.12 Configuration Stuff

Various build mechanisms are used in the [un*x](#) world: simple/complex/nested Makefiles, crazy batch files, `autoconf`, `Imakefiles`, ...

Anyway you should get all available [un*x](#) tools which are already ported, many of them being from GNU. More details can be found at the description of [autoconf](#) and in the [tools section](#).

1.12.1 Makefiles

In my opinion Makefiles are the best build mechanism. Up to obscure internal rules of the Make utils it is a clean and obvious approach and also it's very easy to learn.

- Get a version of GNU make from [OS/2 software sites](#). Watch out for embedded shell commands!
- If you use [XFree86OS/2](#) there's a `make.cmd` in your `%PATH%` which is helpful to some extent and presents a problem in some other cases. You may call the underlying binary executable by `x11make.exe`. This batch file defines `SHELL=""` and may do other things (just currently it doesn't ...).
- GNU make offers many useful features which are beyond the common standard (as defined by [POSIX.1](#), ... ?) for make on [un*x](#). However since GNU make is very portable and widely available you may try to use them anyway.
- To learn about converting `sh`-syntax to `cmd.exe`-syntax refer to the section which discusses [sh vs. cmd.exe](#) issues. Here's just another example, you may convert

```
for i in $(subdirs); do (cd $i; $(MAKE) bar); done
```

to

```
for %i in ($(subdirs)) do (cd %i && $(MAKE) -f Makefile.os2 & cd ..)
```

- Figure out which shell is used, especially if the `SHELL` variable is set explicitly! If the Makfile contains `sh` commands, you may have to add a missing `SHELL=/bin/sh` on your own.

The un*x to OS/2-EMX Porting FAQ

- Have a couple of different make versions around!

Below is an example of some sophisticated feature of GNU Make that x11make does not properly handle while some GNU make 3.76 from [OS/2 software sites](#) does:

```
VERSION := $(shell grep -e "^VERSION=" $(VFILE) | sed -e "s/VERSION=//")
```

1.12.2 Batch Files

The term "batch file" here refers to scripts written for an un*x-like shell (see below). Those shells scan the first line of an executable batch file for the interpreter being required (similar to EXTPROC which cmd.exe recognizes) while you can also write some commands for the current shell without that header:

```
#!/bin/sh
```

Some comments on shell scripts:

- Try to run them in an un*x-like shell like ash, bash, csh, ksh, tcsh, zsh, ...
- To debug shell scripts add this line near the beginning (or the interesting section)
set -x
- Other script language interpreters may be found in [according section](#) of this FAQ.

1.12.3 auto* tools

[GNU autoconf](#) is a tool (set) which produces configuration and Makefiles to build software in different environments. Given the software is properly adapted and autoconf has support for that platform it will then automatically detect the available tools and programming environment (libraries, headers, etc.).

1.12.3.1 autoconf

Source distributions of packages usually include a created "config.h" header which specifies the locally available features via preprocessor clauses like

```
/* Define if you have the "useless.h" header */
#undef HAVE_USELESS_H

/* Define if your libc has the strange() function */
#define HAVE_STRANGE 1
```

For more details on this read the [short introduction into the auto* business](#) written by *Ian Lance Taylor*.

Some OS/2-specific comments are helpful. An important requirement is probably experience

The un*x to OS/2-EMX Porting FAQ

with that stuff on a real un*x-platform, otherwise you'll struggle way too often.

- In case there's a `configure.in` file available you should try one of the existing autoconf ports:
 - ◆ [Version 2.13 \(1.4\)](#)
 - ◆ Version 2.5x (1.5) from [OS/2 software sites](#)
- Often it's used in packages which also make use of [automake](#) and [libtool](#).
- Check out the [list of standard tools](#).
- Ensure that the supplied `configure.in` is compatible to the current autoconf port (e.g. look for `AC_PREREQ()` and no fancy libtool stuff (especially dynamic or shared libraries) is being done. Then things should be as trivial as

```
autoconf && configure && make
```

The default autoconf port lacks a "-Zexe" for plain EMX and such things, but that shouldn't be real problems.

- If you don't have the `configure.in` source try out the "[convert_configure](#)" or "[configure-fixer](#)" scripts from I. Zakharevich. Of course you can also run the "configure" script unmodified, which will at least run through if it's old and simple enough. You may have to fix the resulting files (e.g. `config.h` and `Makefile` manually). If `configure` just stops at some questionable code fragment you might also just drop that stuff. Often even a run with lots of wrong results produces a better set of output files than a non-expert might create ...

1.12.3.2 automake

[automake](#) is a companion to [autoconf](#) and OS/2 ports are available at the same places usually. A project might start from an even simpler set of input files which will then be processed by automake and autoconf to end up with a configure script which will finally create all Makefiles and configuration files. automake itself creates a `Makefile.in` out of a `Makefile.am`. In general all files `configure` should process have to carry the `.in` postfix.

1.12.3.3 libtool

[GNU libtool](#) is being used to build libraries. Usually only for shared libs, since producing static ones should be quite simple (and standard).

- There's an OS/2 port of version 1.2d (see [below](#)).
- Doesn't work for DLLs on OS/2, or at least not in a real useful way.
- Maintaining DLLs which entry points are explicitly using ordinals is a pain. There's a tool (currently part of the [LessTif for OS/2](#) distribution) which does the job for you: `mkdef.cmd`, a powerful REXX program, reads in an old `.def` file, remembers all entries and their corresponding ordinals and while keeping them it assigns unique new ones. The list of the currently available symbols may be automatically created by `emxexp`.
- `link386` has a (documented) commandline flag `/OLD` which should somehow

The un*x to OS/2-EMX Porting FAQ

correspond to a feature of automatic matching ordinals with an older existing DLL, but I never got it to work.

- There are other tools which do help the user to create a DLL without having to write a linker definition file manually.
 - ◆ `dllar.cmd` from [pgcc](#)
 - ◆ "dynamic" script packages from I. Zakharevich (look for "dynamic*.zip" archives)

1.12.4 Imakefile

imake and related tools like *xmkmf* ship with X11 distributions. It is used for X11 software only. Run *xmkmf* (*mxmkmf* exists for LessTif systems; the OS/2 port lacks this so far) to create a Makefile from an input *Imakefile*.

- Often a sequence like

```
xmkmf
make Makefiles
make
```

is required for larger projects.

- Unfortunately an Imakefile may also contain non-portable shell commands.
- To address configuration issues that can't be resolved by using the "knowledge base" of the X11 installation the user sometimes has to adjust configuration files which are then included by the Imakefile.
- Watch out for explicit settings of system-dependent variables like `CFLAGS`, `CXXFLAGS`, a common mistake by software authors!

1.13 X11 Specifics

In general you don't need to make changes specific related to the [XFree86OS/2](#) implementation of the [X11](#) standard. It is very close to the level which is supplied for un*x platforms. In turn [XFree86](#) is based on the official releases from the [OpenGroup](#) and therefore conforming to the standards. Similar to the [M*tif](#) case, here the official implementation *is* the standard ...

In fact most problems arising are already covered in the [XFree86OS/2 FAQ](#) and on the related [mailinglist](#).

How to get a single key pressed in an xterm?

This is a real FAQ for [un*x programming](#).

By default many terminals are in a mode which blocks until RETURN is pressed. The major task to be done is therefore setting the terminal in the correct mode. Fortunately there is a standard interface for this defined within [POSIX.1](#).

You should read about `tcsetattr()` and check out the sample code which I provide for the `select()` call or have a look at some other [sample code](#).

How to work around the non-working SIGWINCH?

The un*x to OS/2-EMX Porting FAQ

Have a look at this [termsize library](#)

Is X11 multi-threaded?

XFree86OS/2 does not support the X11 threading concept called XTHREADS. This is not a critical limitation since any valid software has to check (and can easily do) whether this feature is enabled or not. Here's a suitable autoconf macro:

```
AC_TRY_RUN(
[
#include <X11/Intrinsic.h>
int main() {
Boolean brc;
brc=XtToolkitThreadInitialize();
if (True==brc)
    exit(0);
else
    exit(1);
}
],
have_xthreads=yes,
have_xthreads=no,
have_xthreads=dunno)
```

It is important to perform all X11 calls from a single thread. Otherwise you may get errors like:

```
Xlib: unexpected async reply.
```

This may require quite some efforts if you have an application which uses multiple threads to emulate some missing timer APIs like [g/setitimer\(\)](#).

How to debug an X11 application?

See [debugging.html](#) for some general comments. Remember that an X11 application isn't running straight through some `main()` routine (except for the beginning or end perhaps), but most of the time it stays in a loop which waits for events like keyboard or mouse actions. Then it calls the appropriate routines ("callbacks") which have been installed to react upon them.

How to resolve X errors?

If the app `foo` uses the XIntrinsics (usually linked against `Xt.dll`) run `foo -sync` inside a debugger and [set a breakpoint on `exit\(\)`](#). Also check out the [according section in `debugging.html`](#).

What if an (X11) app fails to link due to unresolved symbols?

If the linking process fails, this probably means you didn't specify some library, or the required libraries aren't available on your system (this is nothing X11 specific). Also you could have missed to specify an object file. Error messages indicating this problem contain long lists of undefined symbols, most of which have a name with an identical prefix, such as `Xm`. The leading underscore which gets added to all symbols (at least using EMX/gcc) is omitted here for brevity.

In the following I list some prefixes and the related linker command to select the required library. A concise description is given as well as an entry which tells you from where to get the library. The entries are sorted by the name of the library, not by the exported

The un*x to OS/2-EMX Porting FAQ

symbols.

X11 related libraries

Prefix	Library	Description	Origin
ftk	-lftk	GUI Toolkit (C++)	ftk (OS/2 port)
fl_	-lforms	GUI Toolkit (C)	xforms (OS/2 ports)
gtk_, GTK_	-lgtk	GUI Toolkit (C)	?
Ice	-lICE	Inter-Client Exchange	XFree86 OS/2
gl_	-lMesaGL	OpenGL clone (3d API)	MesaGL (OS/2 ports)
Mrm	-lMrm	Motif Resource Manager	Motif
PEX	-lPEX	PHIGS Extension for X (3d APIset)	XFree86 OS/2
qt_, ...	-lqt	GUI Toolkit (C++)	?
shm	-lshm	Provides Shared Memory	various sources
Smc, Sms	-lSM	Session Management	XFree86 OS/2
Uil	-lUil	User Interface Library	Motif
X	-lX11	Standard X11 core routines	XFree86 OS/2
Xau	-lXau	X authority database routines	XFree86 OS/2
Xaw	-lXaw	Athena widgets	XFree86 OS/2 (?) (also check OS/2 ports)
Xbae	-lXbae	Matrix Widget Set	Xbae (LessTif)
Xdmcp	-lXdmcp	X Display Manager Control Protocol	XFree86 OS/2
DPMS	-lXdpms	X Display Power Management (DPMS) Extension	XFree86 OS/2
Xdbe, Xext, XShape	-lXext	X Extensions	XFree86 OS/2
Xie	-lXie	X Input Extensions	XFree86 OS/2
Xlt	-lXlt	Widget Set (Motif Extensions)	Xlt (LessTif)
Xm	-lXm	Motif (GUI toolkit)	Motif
Xmu, XEditRes	-lXmu	Miscellaneous X utility functions	XFree86 OS/2
Xp	-lXp	X Print Server	XFree86 OS/2
Xpm, xpm	-lXpm	Bitmap/Pixmap Routines	XFree86 OS/2, ports
XScreenSaver	-lXss	ScreenSaver Support	XFree86 OS/2
Xt	-lXt	X Toolkit Intrinsic	XFree86 OS/2
Tcl_	-lxtcl	Tk (GUI for Tcl)	ports
Tk_	-lxtk	Tcl to X11 interface	ports

The un*x to OS/2-EMX Porting FAQ

XTest, XRecord	-lXtst	Test Extensions	XFree86 OS/2
----------------	--------	-----------------	------------------------------

You may locate symbols which aren't listed here by running `emxexp` on the libraries in `%X11ROOT%/XFree86/lib`. Obviously the library name tends to be the same cryptic letter combination as the prefix you are looking for. Libraries marked as C++ -bindings have mangled names, i.e. a complex mixture between the plain name of the function/variable/method and its signature (if it's a routine). Use `nm -C foo.o` to see the demangled names.

The order in which you have to link these libs is partially fixed:

```
-lUil -lMrm -lXm -lXt -lXmu -lXp -lXext -lXpm -lSM -lICE -lX11
```

Requirement is that a library which depends on another one is specified before that other one. Of course not all of these libraries are always necessary, but in some cases even more libraries may be required. The socket library (`-lsocket`) is another candidate here. It provides e.g. `gethostname()` or `connect()`.

How to link an X11 application statically?

If you want to debug an X11 application and need to deal with X Errors (see [debugging.html](#) and [debugging_os2.html](#)) you may consider to link the X11 libraries statically as well (though you don't gain much here).

To do so you have to get the static libraries package from the [official distribution](#). Then replace all link requests `-lfoo` by `-lfoo_s`. Switching one library from dynamic to static linkage might force you to do the same with others as well! However you may try to link only those statically which are required to satisfy the linker (see example below). In addition new libraries might now be required, e.g. `-lsocket` (see "[What if an \(X11\) app fails to link due to unresolved symbols?](#)"). A sample commandline would be:

```
gcc -Zmtd -Zbsd-signals -o bar.exe bar.o -lXt_s -lX11_s -lsocket
```

Be warned though: statically linked X11 apps might cause problems once you update your X/2 system. My instructions here could also be incomplete: I got some strange errors which I could not explain so far ...

And don't forget that applications for [XFree86OS/2](#) demand to be linked via `-Zmtd`.

How to create a "dualmode" application?

When building a *dualmode* application (command line and X11 app in a single executable) you may wish to link to the X11 libraries statically. This way you can distribute a binary to people which don't have [XFree86OS/2](#) installed.

Actually I should better say "stand-alone, dualmode" application, therefore see also "[How to link an X11 application statically?](#)" Many X11 apps feature a commandline interface as well, e.g. to issue a help message upon `-help`. This is (almost) never found within OS/2 PM applications since this didn't fit into IBM's shortminded concept of separating PM and VIO apps unfortunately. This can be done by "morphing" the application. You may try out the [sample code](#) provided by *Darren Abbott*.

The un*x to OS/2-EMX Porting FAQ

How to distinguish between VIO, PM, X11 session?!

Sometimes one need to check programmatically whether a process is running in a VIO, PM or X11 session. The latter is not recognized by native OS/2 or EMX APIs, so we need to look for some hacks...

If some parent process was started from within the X11 session the environment variable [WINDOWID](#) is set to some integer value. A more thorough check seems to be difficult; even if one would climb up the process hierarchy it's not obvious what to check for: e.g. a search for "xterm" wouldn't catch alternatives like "rxvt". One would have to check whether the current process (or one of its parents) belongs to a window on the current display?!

The even more fundamental check whether X11 itself is available (and running) can be based on that code snippet:

```
#include <X11/Xlib.h>
Display *disp=XOpenDisplay(NULL);
```

which should assign a non-NULL value to `disp`.

If you want to make that binary running without X11 to be installed add those two steps before:

1. check for `%X11ROOT%`
2. try to dynamically load `libX11`:
`dlopen(%X11ROOT%/XFree86/lib/X11.dll) /* pseudo code only! */`
and perform that check.

However in general it's better to link the whole application [statically against the X11 libs](#) then.

Which GUI toolkits/widget sets are available?

Dozens of them exist in the un*x world and many of them have been ported already (check out [sites.html](#)). However there are none of the commercial ones available, i.e. those which don't offer sources for free. Mostly used in the X11 world is the industry standard *Motiffi* (developed & owned by the [OpenGroup](#); core part of it is [libXm](#)). Meanwhile you may use the [OpenMotif](#) release on a few platforms, but not on OS/2 (you may however obtain a source license for big money and try to compile for XFree86 OS/2). Instead you may download *LessTif* from www.lesstif.org: it is a free (LGPL'ed) available clone of Motif. It is source compatible, so you can just rebuild sources written for *Motiffi*.

1.14 External Resources

This is mainly a link section. Except references to documents which I've written I refer to OS/2-related information sources.

1.14.1 Related Documents

Some related documents which I have written and still work on. Most of them were initially just excerpts from this porting FAQ. Then I got the idea that they might be useful for other purposes as well ...

- There is an extraneous tutorial from me about basic debugging issues available, which is not operating system specific. See [debugging.html](#).
- Some system-specific issues might be worthwhile to mention for OS/2-EMX however. See [debugging_os2.html](#).
- Information about printing from non-PM applications is collected within [printing.html](#).
- A short summary about Regular Expressions is in [regex.html](#).
- Sites carrying ported and native OS/2 software are listed in [sites.html](#).

1.14.2 Online Resources

There are many big, useless link lists out there. I try to collect real helpful ones, i.e. canonical resources as well as sites which have real unique offers.

- General Info, Standards
 - ◆ Check out [Standards.html](#) for lots of valuable references!
- OS/2 Software, Ports
 - ◆ [libext project](#)
[libext](#) or [p*six/2 project](#) offers a libc which makes porting very simple!
 - ◆ XFree86 OS/2: [Free X11 Server and libraries](#)
 - ◆ Many site carrying ports are listed on [sites.html](#)
- OS/2-specific Informations
 - ◆ [Installing XFree86OS/2 and an "UNIXish environment"](#) from C. Hennecke
 - ◆ ["Getting Started with EMX/GCC"](#) (an EDM/2 article)
 - ◆ [DevCon](#)
The IBM DeveloperToolbox (former "Developers Connection"=DevCon) offers among other valuable stuff the toolkit docs including [xpg4ref.inf](#), the reference for the portable C API.
 - ◆ [EDM/2](#)
Electronic developer magazine for OS/2
 - ◆ [TT's OS/2 Programming Page](#)
Timur Tabi's site offers many useful links and information
 - ◆ [The OS/2 API Project](#) (discontinued and incomplete, but free!)
 - ◆ [IBM Redbooks online](#)

The un*x to OS/2-EMX Porting FAQ

- ◆ [IBM pubs online](#)
- Misc
 - ◆ [GNU project](#)
those famous guys with a kinky definition of "free software".
Home of the *virulent* license types *GPL* (AKA "General Public Virus") and *LGPL*
 - ◆ Info about the "*OpenSource*" term.
Better avoid it if you can!
 - ◆ [Introduction](#) to various license types
- Mailing lists
 - emx* (emxATxfreeos2.dyndns.org)
[maintenance](#), [archive](#)
 - XFree86 OS/2* (xfreeos2ATxfreeos2.dyndns.org)
[maintenance](#), [archive](#)
 - POSIX/2, libExt project* (posix2ATxfreeos2.dyndns.org)
[maintenance](#), [archive](#)

Send Feedback to Alexander Mai <amai AT lesstif DOTT org>

Last modified on [20031112](#)

2 Printing from non-PM apps

Printing from non-PM apps is kind of difficult on OS/2, at least it's not really well supported. Since all ported applications from un*x lack any support for PM-based printing I try to explain some useful workarounds and solutions. Unfortunately I'm no real expert, you are welcome to supply e.g. native code to access the printer queues, etc.

Major parts of this doc are taken from the *INSTALL.OS2* document from [LyX](#). Feel free to enhance this doc ...!

2.1 Direct Printing

The most simple solution is printing to a Postscript file and then use gvpd for printing. If you have installed "printlp.exe" of "gnugroff.zip" from [OS/2 software sites](#), you could try to directly print to the local OS/2 spooler: Excerpt from README.OS2 as supplied with 'gnugroff.zip':

New printlp and printdvi programs: Printlp captures all stdin input and spools it to "lpt1" (default) or the output device set in the environment variable LP_DEVICE. Printdvi captures all stdin input into a temporary file and calls the command "v" with the name as argument (v is the viewer in emTeX) or the command in the environment variable DVI_PRINT_COMMAND. If the command contains a %s, this will be replaced by the temporary file name.

I recommend to add to config.sys

```
SET DVI_PRINT_COMMAND=VP
```

or "printlp" for a Postscript-capable printer or if you already configured a postscript filter for your program. A different solution to try is this one from Marcus von Cube (for xephem): Create a small REXX file:

```
/* xprint.cmd */
parse arg file
file = translate( file, "\", "/" )
'@print /d:lpt2 /b' file
```

(you may want to change "print /d:lpt2 /b" to "printdvi").

2.2 LPR, LPD

Warp 3 with common extensions and all versions above contain helpful tools for our purposes: "lpr" and "lpd". If your %ETC%\INETD.LST contains the following line:

```
printer tcp lpd
```

and inetd is running, you should be able to use lpr:

```
lpr -s localhost -p lpt2 -b <filename>
```

The un*x to OS/2-EMX Porting FAQ

(it might be necessary to set up a *loopback interface*, see networking FAQs or XFree86OS/2 installation docs how to do this).

It accepts "/" as a path separator. For convenience you may create a batch file which may inherit the permanent flags. e.g. a "lp.cmd" which looks like

```
lpr -s localhost -p MyPrinter %1 %2 %3 %4 %5 %6 %$7 %8 %9
```

where we assume you have a printer queue named "MyPrinter" on your machine. Knowledge of the physical names of your printer queues is required for this approach. You can retrieve them by looking looking at the "View" page of the according printer object on the WPS or in the "\SPOOL" directory on your system's partition.

A superior way is to use the more complex OS/2 API. Check out e.g. `SpLEnumQueue()`. This API allows to retrieve the default queue. The other `SpL*` interfaces allow a full job control even without any PM-GUI interaction.

2.3 Wrappers, Utils

un*x ports require at least two features to make them work as being on an un*x system:

1. a Postscript interpreter or printer, since [Postscript](#) is *the* standard printing format
2. equivalents to "lp", "lpr" utilities

Several utils exist to help integrating such tools in your OS/2 system.

- GhostScript is the only Postscript interpreter that you could use. It can be found at <http://www.cs.wisc.edu/~ghost/>. However it comes with a non-advisable OS/2 binary set. Better get one with X11 support from [J. Sawataishi's website](#) or from [OS/2 software sites](#)
- A more convenient example, actually a poor man's un*x "lpr" clone can be retrieved as a small [ZIP archive](#).
- A Simple lpd-like daemon is available: [pmlpd102.zip](#)
Citing the README:

This program looks for a specified file to be created. When the file exists, a given process will be started. The program waits until the process has ended, and then it will erase the file and wait for a new one. It was written as a daemon for GhostScript.

- Check out "lportd" or tools like "printmon" (from [OS/2 software sites](#).)
- To deal with Postscript documents you can use the *psutils* (e.g. from from [OS/2 software sites](#)). They offer all kinds of manipulation including printing 2 or n logical pages per sheet, creating booklets or printing only selected pages of a document. All this can be done from command line.

Last modified on 20020105

3 Debugging

This is a small summary for beginners on *how to debug a program* (experts in turn should read it and then send me corrections&additions :-)

Since this is a complex issue I just try to explain the first steps of locating errors which correspond to a "critical" signal, like SIGSEGV: this indicates a problem that is so crucial that the operating system notices about it and forces the application to quit. More subtle problems are not caught this way ...

Even non-programmers should be able to handle this, however one should be roughly familiar with the operating system being used.

3.1 Tools

Ideally you should have the *source code* for the application and all involved libraries. However I assume that standard libraries don't easily crash, but give some "reasonable" feedback so I concentrate on bugs within the non-standard parts. An essential prerequisite is a *debugger*, a tool which allows you to run an application step by step and examine the internal data. On many systems the GNU debugger (gdb) is present, however it may be inferior to commercial ones (like dbx, ladebug, etc.).

Two basic kinds of debugging applications are available:

- Source code checker. The most famous example being *lint*. On the "freeware side of life" check out [splint](#) (former "lclint")!
- Others, to work with the existing executable, might be runtime or even post-mortem analysis. Examples for such are [malloc debugging](#) libraries and profiling tools.

3.2 Procedure

I try to outline the procedure to be followed and specify some details within the brackets [] on a fictitious example written in C which assumes gcc (GNU C compiler) and gdb are present to create and debug an executable "foo.exe". Note that the short versions of the gdb command s are used here; these are less descriptive than the full command strings, but I'm too lazy to write them here ...

1. Prepare a set of *debuggable* binaries (i.e. libraries and executables).
This involves specifying a flag to the compiler (and linker) [-g] upon compiling and linking. Otherwise the compiler/linker may not store valuable information which shows the corresponding source to object code and vice versa.

```
gcc -o foo.exe -O0 -g foo.c
```

2. You should *disable optimization* when compiling [-O0 is used for many compilers]!
Note that this may have apparently subtle effects:
 - ◆ Bugs may come and go due to internal flaws of the compiler when changing the optimization level.

The un*x to OS/2-EMX Porting FAQ

- ◆ Bugs may also be spurious due to common optimization procedures:
e.g. if your code contains a statement which would trigger an error on runtime it might just disappear when turning on optimization (e.g. by removal of unused code). The sample below is a candidate for that phenomena:

```
void foo(void) {
int dummy; /* The var "dummy" is only referenced once in the code,
           namely the assignment below */

/* try to calculate square root of (-1): */
dummy = sqrt(-1); /* This should trigger an error due to
                  an illegal argument */
}
```

- ◆ gcc has a shortcoming in its design: completely disabling optimization (-O0) limits its capability to detect potential problems in the source, so it won't issue warnings accordingly. Read the gcc manual!
3. Run the **debugger** on the executable [gdb foo.exe]
 4. Optionally set a **breakpoint** on `exit()` [b exit]
The debugger will stop execution when the given location (might relate to a line in the source code, a function, etc.) is being reached which has been assigned to a breakpoint. Then the command prompt is presented again.
Check below for more information about `exit()`. Stopping there is not always necessary, but required when looking for [X protocol errors](#).
 5. Actually **run the program** [r].
Take care here if debugging [X protocol errors](#).
In case you need to specify **commandline arguments** you can do this before running the code either by an explicit call [set args bar] or "implicitly" [r bar].
 6. Now try to **reproduce the bug** in the *most simple fashion*, i.e. minimize the number of "actions" like keystrokes and mouseclicks, use most simple input data file, etc.
Remember exactly (even better write down) what you do here!
 7. The debugger commandline will tell you technically what has happened (e.g. a segmentation fault, SIGSEGV, has been caught). Now the important thing is to locate it in detail.
Produce a **stack trace** [where]. This will show the location within the program where the crash happened.
Attach a copy of this output to your bug report!
 8. Given the crash has happened in a function for which you have the source code (I assume so) you can now **list the source code**, i.e. the "bogus" line [l].

The next step would be examining further details (stack itself, arguments to current function, invalid data, etc.).

3.3 Advanced Issues

Though I can not explain further procedures in this recipe-like style, I want to give some "well-known" hints & ideas and address some standard tasks to be done.

3.3.1 Resolving X Errors

An *X protocol error* happens when some improper request was sent to the X server. It can happen when programming directly the *Xlib*, or if the used toolkit issues such a broken request. So if you're not actually programming low-level or working on an X11-based GUI library this error won't be your fault. This section is also useful if you're a newbie to debugging X11 programs in general, it's not only limited to X protocol errors only.

Within the [debugging procedure](#) outlined above you should check on how to set a breakpoint on `exit()`. Check whether the application uses the X Intrinsic (`Xt`) library (e.g. run `ldd foo.exe` outside `gdb` within a shell). If it's listed there run the application with the `-sync` option [`!r -sync`]. If not (pure *Xlib*) set the global variable `_Xdebug` from within a debugger or even within the source code near the begin of `main()`.

Note that this may also change the "location" and/or "appearance" of the bug or even cause it to disappear!

Alternatively one may trigger on the interfaces which the X11 libraries call if there's a problem which they can detect: You may try setting breakpoint to the *Xlib* calls `_XDefaultError()`, `_XError()`, `_XIOError()` and Intrinsic lib calls `XtError()`, `XtWarning()`.

If things go wrong upon system/libc calls from within that libraries, those interfaces won't be called so you rely on your libc only. If your breakpoint on `exit()` doesn't help when an X11 application crashes, check out the [alternatives](#).

An important issue while debugging X applications is that one can very easily lock up the display so that mouse and keyboard may no longer accept input. To avoid these issues there are some workarounds:

- Run the debugger on a console (or PM window/fullscreen session if on OS/2).
- Run the application in `Xvfb` (**X Virtual Framebuffer**), `Xnest` or on another (local) display (e.g. `" :1 "`). While the first one is a good idea only for code which may be examined non-interactively (e.g. check for memory-handling, profiling, etc.) the latter two alternatives are not only substitutes but almost better than running on the current X server. Especially `Xnest` is helpful if you need to take care of data like Atoms being stored in the X server.

Unfortunately both tools are not always available.

Further details on debugging X11 apps can be found at these valuable writings:

- www.openmotif.org/tnt/#Debug_Breakpoints
- www.rahul.net/kenton/perrors.html

3.3.2 Well-known Tricks

Everyone should know those tricks ...

The un*x to OS/2-EMX Porting FAQ

- Use a malloc debugging library like [dmalloc](#), [dbmalloc](#) or [efence](#). Probably the majority of all bugs is due to improper memory handling. Also they might help to detect *memory leaks*.
- Discover more warning flags of your compiler! Beyond some basic problems a verbose compiler might tell you about more subtle problems.
- Ensure the compiler is in an ANSI conforming mode and not just K&R or something else.
- Try compiling/building on a different operating system/architecture with a different compiler.
- If not disabled (see "man ulimit") programs often produce a core dump when they crash. The resulting data file is an image of the memory when the app was stopped. You can perform a post-mortem crash analysis with this file. Run the debugger with the executable and the core dump as arguments [`gdb foo.exe core`] and proceed as explained [above](#).
- `exit()` is a function which is being called usually upon program termination. Then a lot of internal clean up is done. Setting a breakpoint there might also help when you're looking for memory leaks. When debugging C++ apps crashing upon `exit()` look for destructors being called there.
- Sometimes an application crashes but you can't easily detect where and your breakpoint on `exit()` doesn't help. Then you should check other "legal" procedures within `libc` a program may call on exit, including `abort()` and `_exit()`.
- `main()` has a similar meaning as `exit()`: a lot of things will be performed which the simple-minded programmer might not be aware of. Actually as opposed to `exit()` things will happen *before* the program's `main()` is being called. So your program might crash due to all kinds of improper initialization (e.g. variable assignments) and all C++ constructors being called.

3.4 Compile Time Problems

You may even have problems to get some code compiled. In this section I will again list a couple of ideas that have helped me at least once to get things resolved. Some of them may depend on the compiler being used however.

- Macros obscure the effective code. Try `gcc -E` to get the output as it's being passed from the preprocessor (cpp) to the compiler.
- The former command does not tell you which macros are actually being set. Try `gcc -E -dD` and `man cc` for that.
- Compilers may choke on source code which is not in the native text (DOS vs. un*x) format or includes some special characters. While the first are broken legacy compilers the latter is quite reasonable behavior ...
- Often macros shield declarations and definitions in the system headers. If you're able to locate a declaration/definition of a symbol which misses in your compilation in the system headers try to check with the man page of your compiler. (obviously the direct approach is reading the headers, but the used macros may be deeply nested ...)

3.5 Portability Issues

Nowadays *portability* is very important. Writing clean code saves you a lot of time and also increases the chance to get it built easily on 64-bit machines as well.

So in case you run across problems within an application it might be that the problem is not a source code which is totally broken, but it has just been written & tested in a single environment. From this point of view *portability* is a crucial thing to reduce the amount of useless debugging procedures.

In the following I briefly mention some famous portability issues.

3.5.1 Language Level

- Never use pre-ANSI interfaces, e.g. those included within the headers `memory.h`, `<strings.h>` or `<varargs.h>`!
- Don't use compiler or preprocessor extensions!
Famous examples are the GNU extensions `__FUNCTION__` and `typeof()`.
- Check whether compiler switches cause a different behavior of the executable. An example are the compilers on machines based on the alpha processor (AXP) which require `-ieee` or `-mieee` to get applications to work which rely on IEEE conformance and proper handling of numerical exceptions.

3.5.2 "Bitness"

- The famous *byte order* issue: most widely used are `LITTLE_ENDIAN` ("1234", on i386) and `BIG_ENDIAN` ("4321").
- Do not assume `char` to be signed or unsigned!
- Do not assume `sizeof(int)` equals `sizeof(long)`!
- Do not assume `sizeof(void *)` equals `sizeof(int)`!
i.e. don't assign pointer values to `int`
- Do not assume that the result type of the `sizeof` is `int`!
Its type (an unsigned integer type) is `size_t` which is defined in the `<stddef.h>` header.
- *Always* use full prototypes and make your function calls to fulfill the specified signature with explicit type casting.
- The former rule is important when using *varargs* interfaces (variable number of arguments), e.g. `va_start()` from `<stdarg.h>` or from other libraries like the `XtVa*()` interfaces from the X Intrinsics library. Usually they need a `NULL` pointer to indicate the end of their argument list. Implicit conversion from 0 won't work, so you have to use `NULL` (though even a strange defined *NULL* may cause trouble).
- Even in the beginning of the 200x decade using C9x-features is not a good idea: at least the full set of features is rarely implemented and in even more rare occasions you will actually find such a compiler on your target machine ...

3.6 Bug Reporting

If you're going to write a proper bug report you need to consider of couple of things:

First tell exactly which *version of the code* you are using, give the exact version of that distribution or the CVS checkout date. You should specify all libraries which are involved. Run `ldd foo` to see all shared libraries linked to the executable. (Note that `ldd` is not a standard tool, it may have different names on other platforms or even do not exist on your installation!)

If the error happens while compiling/linking always give the full command line, perhaps even the complete output of some "make" command.

In addition you need to fully *specify your system*. "`uname -a`" should give you the details about your operating system as well as the basic hardware (CPU architecture).

Happy debugging!

Last modified on 20020102

4 Debugging on OS/2

Previously I wrote some notes about [basic debugging issues](#). Here I want to briefly mention some OS/2 and EMX/gcc specifics. Peculiarities of debugging PM apps are not covered here.

4.1 Tools and Helpers

For good reasons I only refer to "no cost" products here ...

- There are some *debuggers* available for free:
 - ◆ *gdb* (pmgdb) (for a.out objects; part of EMX)
GNU software; available for most platforms
 - ◆ *sd386* ([OS/2 software sites](#)) (for omf objects)
from [IBM](#)
- Resolving *segmentation faults* (SIGSEGV):
Watch out for debugging implementations of `malloc()` like *dmalloc*.
dbmalloc
is available from [OS/2 software sites](#). It's rather old now (≥ 7 years), but still builds and the supplied test examples work.
If you're going for a quick check it is sufficient just to link against this library:

```
gcc -Zmt -Zexe -o foo foo.o -lbar -ldbmalloc
```


But often you will prefer to get a more helpful output than the one supplied in this case and add `#include <dbmalloc.h>` to all your source files (actually you don't need to do so for *all*, but it is a good idea ...) and rebuild.
dmalloc
I'm offering a [build for OS/2 EMX](#). Usage is quite similar to `dbmalloc`.
libefence
a well-known one on un*x, isn't available on OS/2.
ccmalloc
nice one for i86 linux, isn't available on OS/2.
- Operating System/2 API Trace (os2trace.zip on [OS/2 software sites](#)):
Enables, customizes, controls and summarizes the tracing of OS/2 APIs imported by a 16-bit or 32-bit executable file without affecting its source code or requiring recompiling or relinking.

4.2 Debugging Basics

Though called "basic" the following methods may not be obvious even if you already have some experience with debugging in general.

- To set a *hard-coded* breakpoint (see [SIGTRAP](#)) within your application you may use this macro:

The un*x to OS/2-EMX Porting FAQ

```
#define BREAKPOINT __asm__( "int3" );
```

- **core** dumps are images of the current process (mainly the memory) written to the disk. On EMX they're only available if using *a.out* objects, and so could afterwards be debugged using GDB. Core dumps are even available upon request, see `_core()`.
- Don't mix OS/2' and EMX' memory handling!
- **Breakpoint** on `exit()`:

If an executable is linked against `emxlibc.m.dll` (or `emxlibcs.dll`; I will use the term "`emxlibc*.dll`" in the following) `exit()` and other libc functions (also those related to `exit()` like `abort()` and `_exit()`) are not known symbols to *gdb*. Unfortunately this usually happens when building X11 apps (see [XFree86 OS/2 FAQ](#)).

`exit()` is located in `emxlibc*.dll`, which was linked from omf objects using *LINK386*. *gdb* is unable to resolve symbols by name from DLLs of these kind. Therefore it doesn't know the address of `exit()` and cannot set a breakpoint on it. Look up its offset in `\emx\etc\emxlibc*.map`, e.g.

```
0001:0000DB78 exit exit
```

Use `set show-dlls` to make *gdb* stop upon accessing the DLL for the first time and set a specific breakpoint on that specific DLL by using `dll-break emxlibc.m`. From that output, e.g.

```
[Load DLL: E:\PROGRAM\EMX\DLL\EMXLIBCM.DLL]
[.text: 0x1db80000 - 0x1dba9a80]
[.data: 0x187b0000 - 0x187b6060]
[.bss: 0x187b6060 - 0x187b97c0]
```

extract the `.text` (i.e. executable code) base address. Then set the breakpoint on the address calculated as the sum of base and offset:

```
b *(0x1db80000+0xdb78)
```

The EMX docs claim that "due to a bug in OS/2 the breakpoint will apply to *all* programs using `emxlibc*.dll`". I couldn't verify this with recent versions of OS/2 ...

- **Resolving X Errors**

See [debugging.html](#) for some general comments on this issue.

Since setting breakpoints in the X11 libs instead of libc (`emxlibc*.dll`) requires to link a set of debuggable [X11 libraries statically](#) and so doesn't help much with XFree86 OS/2/EMX usually.

4.3 Misc Hints

This is intended to be a collection of ideas if you run out of them while trying to resolve a problem. Here I could really need more input from other developers!

- Subtle segmentation faults are sometimes triggered by running out of stack space. An endless recursion is a good candidate for this. Or the stack size was given a too small

The un*x to OS/2-EMX Porting FAQ

value (see `-Zstack` option of `gcc`). Since this can not be fixed on OS/2 during runtime (but only while linking the application) just make it big enough for any possible situation. OTOH setting it too big may end up with more subtle errors, like `sys1059` when trying to start an application ...

- Another candidate is usage of stale pointers: references to memory/variables of storage class `auto`. Hard to debug, since few tools will tell you when trying to free a reference to such a pointer.
- If some problems happen with input being read (binary data like images, or text files as well) check whether the file (socket, pipe) is being read in the correct mode (see `fopen()`, `-Zbin-files`). You may have to write explicit code to read in text files which may be in either [DOS or un*x format](#).
- Make sure a program which accesses DLLs (via `import libs` or dynamically) loads the correct versions of those libraries. You may use `ldd foo.exe` to see the shared libs of that executable (check the [porting FAQ](#) for that command).
- Applications which depend on signals (including usage as a timer, for data acquisition, animations) and which don't work properly might suffer from using the wrong signal model (see [Porting FAQ](#)).
- Sometimes one forgets that `fork()` doesn't work in an executable based on `omf-objects`/linked with `link386`.
- If you believe to have discovered a bug in your current EMX/gcc try the [alternative versions](#).

4.4 Other Resources

Here I collect references to other information resources which have not been mentioned so far.

- [IBM OS/2 Debugging handbook](#) (INF format)
- More info about *traps*, the native OS/2 error mechanism, can be found in
 - ◆ *The Control Program Guide & Reference*, under Exception Management
 - ◆ an [article from Frank Meilinger](#) and
 - ◆ the [short summary about traps](#) from Steven Grim.
- [except3.zip](#)
Contains sample code using exception handling for debugging purposes.

Last modified on 20020112

5 Regular Expressions

The world of Regular Expressions isn't as simple as it could be: there exist many versions of them: POSIX defines Basic (BRE) and Extended Regular expressions, and in addition every language and tool tries to come up with its own enhanced set (Perl, GNU utils, emacs, ...)

This an incomplete, perhaps not even correct short reference list of some common denominator of Regular Expressions. Unfortunately I have no idea where I got it from.

5.1 The Metacharacters of Regular Expressions

Character	Meaning
\	Turn off the special meaning of the next character
^	Indicate the beginning of the line
\$	Indicate the end of the line
.	Any single character
*	Any number of occurrences of the previous character
[]	Any one of the characters inside the square brackets
[^]	Any one of the characters not inside the square brackets
\(\)	Tagged regular expression

5.2 The Syntax of Regular Expressions

The syntax of regular expression is basically the combination of the metacharacters and the regular characters. For example:

<code>^My</code>	matches line starting with the word "My"
<code>finally\$</code>	matches ending with the word "finally"
<code>^I am a boy\$</code>	matches the whole line of the sentence "I am a boy"
<code>\\$10</code>	matches "\$10", meaning of \$ is turned off
<code>t.e</code>	matches "t" followed by any one character and then an "e"
<code>my*</code>	matches "m" and followed by any number of occurrences of "y"
<code>th.*</code>	matches "th" and followed by any characters including no character
<code>[124]</code>	matches one of "1" or "2" or "4"
<code>[12-46]</code>	matches one of "1" or "2" or "3" or "4" or "6". Expanded as "2-4", not twelve to fortysix.
<code>[^abc]</code>	matches any one except "a" and "b" and "c"

Last modified on 20010115

6 Ported Software Sites

This is a list of interesting sites where you can find software for OS/2, including X11 applications.

6.1 Collections

Sites which just collect other people's software:

- UNIXOS/2
 - ◆ [Website](#)
 - ◆ [FTP archive](#)
- XFree86 OS/2's outdated list of ported applications on <ftp://ftp.xfreeos2.dyndns.org/pub/misc/xfreeos2/oldarchive/html/xf86ported.html>
- New [Porting site](#) from Mateusz Latusek

6.2 Developers' sites

Homepages of individuals or groups which offer real unique content:

- Hung-Chi Chu's site
<http://r350.ee.ntu.edu.tw/~hcchu/os2/ports>
Ported libs; port of autoconf&Co., shm, GNU gettext, lots of ported software
Meanwhile offline, check the [unixos2](#) site instead.
- Website of [J. Sawataishi](#)
A lot of ported software
- Website of [Kiyosuke Tokoro](#)
Miscellaneous ports/programs
- Ports at pluto.spaceports.com
- Website of [Alexander Mai](#)
Miscellaneous ported software, Porting FAQ, ...

Last modified on 20020103

7 Standards & Docs

This is a collection of references to essential documentation about programming issues and API standards. Especially it focuses on public available resources, namely stuff which may legally be retrieved from the internet.

7.1 Official Standard Documents

Often the *official* standard documents are rather expensive, even if purchased without a hardcopy. So the clever people know where the (final) drafts can be found ... (and they are aware that those are *not* the "real" standard documents!)

7.1.1 C

- [ANSI C standard \(ISO WG 14\)](#)
 - ◆ [ANSI C89 rationale \(HTML version\)](#)
 - ◆ [ANSI C99 rationale \(draft\)](#) (also as a [PDF](#), and an [older version](#))
 - ◆ [ANSI C99 standard \(draft\)](#) (also [http](#), in [html](#) and in [PDF format](#))

7.1.2 C++

- [ANSI C++ standard \(ISO WG21\)](#)
 - ◆ [Standard \(draft, December 1996\)](#)
 - ◆ [dto](#) (HTML'ized online version)

7.1.3 Fortran

- [Fortran 77 standard document](#)
- [Overview about ISO Fortran standard drafts](#)
 - ◆ [Fortran 95 drafts](#)
 - ◆ [Fortran 200x drafts \(here as well\)](#)

7.1.4 un*x & related API References

- ***POSIX***

Standard for an operating systems environment, including tools and libc.
There are no legal copies available for free

 - ◆ [Austin Group](#) standard drafts:
X/Open and POSIX join now to create a unique definition. You can get the drafts for free if you register (at no cost). Latest is *draft 7*, the final one!
There's an HTML version of the [final standard online](#).
 - ◆ [older drafts](#)
- ***UNIX***

The un*x to OS/2-EMX Porting FAQ

- ◆ [Unix man pages \(Single UNIX Specification\)](#)
- ◆ [API, standards reference table \(older version\)](#)

7.1.5 Library References

- *X11*
 - ◆ [X11 home](#)
 - ◆ [XFree86](#)
A free X server (including all related programming stuff)
 - ◆ [X11 and related references \(man pages\)](#)
 - ◆ [Info about Imake](#)
- *Motiffi*
 - ◆ [Motif home](#)
 - ◆ [Open Motif Docs](#)
 - ◆ [Links to info about M*tif](#)

7.1.6 Misc Programming

- *Math Programming, Numerics*
 - ◆ [Infos about floating point math, IEEE 754, etc.](#)
 - ◆ [Drafts for ISO/IEC 10967-1](#) (standard for language-independent arithmetic (LIA-1))
- *Internet Standards*
 - ◆ [Requests For Comments \(RFC\)](#)
(alternative site)
 - ◆ [RFC drafts](#)
- *UNICODE*
 - ◆ [UNICDE standard documents](#)
- *Binary Formats*
 - ◆ [Object/executable file formats](#)
 - ◆ [List of several file formats](#)

7.2 FAQs

Frequently Asked Questions are often compiled by some volunteers and made available in the net for free. Of course they also contain some more or less good, precise, valuable answers ...

- [C FAQ](#) ([alternative site](#); older version in [HTML](#))
- [C++ FAQ](#)
- [C++ FAQ lite](#)
- [comp.std.c++ frequently asked questions](#)
- [Fortran FAQ](#)
- [Another Fortran FAQ](#)
- [FAQ for un*x programming](#)

7.3 Implementations

Sometimes it might be helpful to see real-world examples of the standards given above. So I also collect some links to implementations. Take care to read the according docs, which will (hopefully) tell you about the conformance with the related standards.

- *Misc un*x Stuff*
 - ◆ Useful [UNIX links](#)
 - ◆ [Information about various un*x flavours](#)
 - ◆ [Porting across un*x flavours](#)
- *BSD Systems*
 - ◆ [FreeBSD](#)
 - ◇ [FreeBSD sources](#)
 - ◇ [FreeBSD sources cross-reference](#)
 - ◆ [NetBSD](#)
 - ◆ [OpenBSD](#)
- *linux*
 - ◆ [Linux source code reference](#)
 - ◆ [Miscellaneous Linux documentations](#)
 - ◆ [Linux man pages](#)
- *Sources, Libraries*
 - ◆ [Freely Distributable LIBM \(fdlibm\)](#)
Missing math stuff can be found here
 - ◆ [cephes](#)
Math library, too

7.4 "Non-Programming" Standards

One also frequently requires non-programming standards, e.g. when writing documentation, etc.

- *SGML* & friends
 - ◆ ["WebSGML"](#)
 - ◆ [Document Style Semantics and Specification Language \(DSSSL\)](#): last draft of ISO standard
 - ◆ [Extensible Markup Language \(XML\)](#) (subset of SGML)
 - ◆ *HTML* & friends
 - ◇ [ISO HTML](#)
 - ◇ [HTML 2.0](#)
 - ◇ [HTML 3.2](#)
 - ◇ [HTML 4.01](#)
 - ◇ [XHTML 1.0](#)
- [Portable Document Format \(PDF\)](#)
 - ◆ [PDF reference](#)
- [Postscript \(PS\)](#)

The un*x to OS/2-EMX Porting FAQ

- ◆ [Postscript Language Reference](#)

7.5 Definitions

The following list is taken from the linux man pages (INTRO(2) linux Programmer's Manual), which is not up-to-date. So I will correct & update it when I find some time ...

SVr4

System V Release 4 Unix, as described in the "Programmer's Reference Manual: Operating System API (Intel processors)" (Prentice-Hall 1992, ISBN 0-13-951294-2)

SVID

System V Interface Definition, as described in "The System V Interface Definition, Fourth Edition", which is available in Postscript format from <ftp://ftp.fpk.novell.com/pub/unix-standards/svid/>
[broken link - try at <ftp.inf.tu-dresden.de> instead]

POSIX.1

IEEE 1003.1-1990 part 1, aka ISO/IEC 9945-1:1990s, aka "IEEE Portable Operating System Interface for Computing Environments", as elucidated in Donald Lewine's "POSIX Programmer's Guide" (O'Reilly & Associates, Inc., 1991, ISBN 0-937175-73-0.)

POSIX.1b

IEEE Std 1003.1b-1993 (POSIX.1b standard) describing real-time facilities for portable operating systems, aka ISO/IEC 9945-1:1996, as elucidated in "Programming for the real world - POSIX.4" by Bill O. Gallmeister (O'Reilly & Associates, Inc. ISBN 1-56592-074-0).

SUS, SUSv2

Single Unix Specification. (Developed by X/Open and The Open Group. See also <http://www.UNIX-systems.org/version2/>.)

4.3BSD/4.4BSD

The 4.3 and 4.4 distributions of Berkeley Unix. 4.4BSD was upward-compatible from 4.3.

V7

Version 7, the ancestral Unix from Bell Labs.

Last modified on 20030424