

Use the Imakefile!
or:
Some mean tricks to get the stuff compiled!

Copyright 1 for the compilation:
Arnd.H.Hanses@rz.ruhr-uni-bochum.de

Towards an UNIX/POSIX to emx-OS/2 Porting Guide

*"Good programmers know what to write.
Great ones know what to rewrite (and reuse)."*
(ERIC S. RAYMOND, IN: "THE CATHEDRAL AND THE BAZAAR")

*Alfieri: > E la fama? <
Gozzi: > E la fame? <
(COMMEDIA DELL'ARTE)*

Informal introduction into aspects of porting from UNIX/POSIX to OS/2 that are closely related to the emx development tools. Teaching ANSI C is not intended, some familiarity with programming in C or C++ is assumed; pointers to tutorials for rank beginners are included.

(Authorship is usually indicated. Please mail me any errors and suggested additions. I am looking forward to adding new chapters written by interested readers. If the author's name is omitted and not evident, I am citing usually an excerpt from the emx documentation. Those hundreds of trademarks are those of their respective owners.)

Contents

I	The Tutorial	9
1	Instead of an Introduction	11
1.1	Why Are the Emx Development Tools Important for Writing Modern Portable Code ?	11
1.1.1	What's emx?	11
1.1.2	What's That Hype About "Code Reuse"?	13
1.2	Where to Start?	14
2	Survive With Emx Tools (cc, make, gdb, emacs)	17
2.1	Let's contrast Emx With FreeBSD.	17
2.1.1	Compiling With cc	17
2.1.2	Common cc Queries and Problems	20
2.1.3	Make	23
2.1.3.1	What's make?	23
2.1.3.2	Example of using make	23
2.1.3.3	More advanced uses of make	24
2.2	Debugging	25
2.2.1	The Debugger	25
2.2.2	Running a program in the debugger	25
2.2.3	Examining a core file	27
2.2.4	Attaching to a running program	28
2.3	Configuring Emacs	30
2.3.1	A sample .emacs file	31
2.3.2	Example A sample .emacs file	31
2.3.3	Extending the Range of Languages Emacs Understands	37
2.4	Further Reading	38

II	Special Problems	41
3	Frequently Used Concepts	43
3.1	What's ' \mathcal{X} '?	44
3.2	The 'official' port of XFree86	45
3.2.1	General Information	45
3.2.2	The XFree86/OS2 Infrastructure	46
3.2.3	Tip: Everblue	46
3.3	And Posix, etc.?	47
4	A Silly Demo and Some Comments	51
4.1	Read the Emx Documentation	51
4.2	Source with Comments	52
4.3	Module Definition File - Executable	55
5	XFree86: Hardware Gets Involved	59
5.1	Devices, \mathcal{X} Servers, Clients	60
5.1.1	'/dev/fastio\$', '/dev/console\$', '/dev/ptyP0\$', etc.	60
5.1.2	Concepts Related to Hardware (vio) Access and Kernel Level API 's	60
5.2	Inter Process (Session) Communication	62
5.3	Emx and device driver programming (hardware devices)	62
6	Building Libraries	65
6.1	Explaining Some Library Concepts	65
6.1.1	Dynamic Link Libraries (DLL's) Versus Static Libraries	65
6.1.1.1	Static Libraries (*.a or *.lib)	65
6.1.1.2	Shared Objects (*.so)	65
6.1.2	Shared and Dynamic Libraries (TOG Spec1170)	66
6.2	Exporting symbols from a DLL (OS/2 OMF)	67
6.3	'dynamic library' and 'shared object' (*.so) versus DLL	67
6.4	Demo	68
6.5	How to Link Your DLL	70
6.6	Library Debugging	70
6.7	Building a C++ DLL	71
6.8	Some Special Tricks for libraries	71

6.8.1	Instead of using preprocessor macros, you can use import libraries	71
6.8.2	Debugging huge DLL's	72
6.8.3	Length limitation for symbol names in OMF-Format	73
6.8.4	Specify the name without the extension	74
6.8.5	C++-DLL's	74
6.8.6	Multiply defined symbols error with C++-DLL's	75
6.8.7	Sys 0008	76
7	Imakefile or configure, that's the question!	77
7.1	A short overview: Autoconf vs. Imakefile (by Hung-Chi Chu <hcchu@r350.ee.ntu.edu.tw>)	77
7.2	Preliminary steps	79
7.2.1	RTF docs	79
7.2.2	Some additions to /XFree86/lib/X11/config/os2.cf and os2.rules	81
7.3	Common initial problems	87
7.3.1	'xmkmf -a' crashes because of 'sh' and 'Rexx' conflicts	87
7.3.2	'make' crashes because of 'sh' and 'Rexx' conflicts	88
7.4	Imakefile examples	89
7.4.1	creating a static library (archive): NormalLibraryTarget(rw, \$(OBJS))	89
7.4.2	creating an executable: ComplexProgramTarget(xart)	90
7.5	Some maybe useful defines	92
8	XFree86/2 and Multi-threading	95
9	Numerical Math	97
9.1	Types, Limits, Conversion.	97
9.2	IEEE Maths.	98
9.3	Useful Math Extensions	100
10	What Shall I Better Change in the Sources?	107
10.1	Globbering	107
10.2	DOS "text" mode, binary reading, etc.	107
10.3	Process management and fork()	108
10.3.1	What is the difference between fork() and spawnvp()?	108

10.3.1.1	<code>fork()</code> and <code>emx</code>	108
10.3.1.2	making it work	109
10.3.2	replacing <code>fork()</code> and <code>execvp()</code> , etc.	109
10.3.3	<code>fork()/exec()</code> and dual way pipe communication	110
10.3.4	Example (C++-Code from modified LyX 1.0.4 sources):	114
10.4	Multithreading instead of <code>fork()/vfork()</code>	120
10.5	Unix file system issues	120
10.6	Sockets	122
10.7	Environment (<code>getenv()</code>)	123
10.8	Miscellanea	124
III	Appendices	125
11	Some Notes on Portability and Coding Style	127

"I have found a cool X utility which I want to port. How should I proceed?"

Normally, with X utilities, you get an Imakefile. Simply proceed as if you were under Unix. Run xmkmf -a to compile. Read the programming section on special issues."

("The XFree86/OS2 FAQ")

Preface

(What should be the aim of an emx-tutorial?)

Since I am not a professional programmer, I am advocating user friendly tools and user friendly documentation. I do not think such helpful things should aim at becoming large or comprehensive, but simple and down-to-earth. It should just enable the first simple steps and help to reduce the beginners problems, thus providing the necessary basis and some pointers so that you learn how to get access to the huge piles of helpful information on the Net and elsewhere. Well, it is time now to make the first step towards a (still epigonous and rudimentary) porting tutorial, compiling in the form of short articles some of the helpful information that's scattered in small bits and pieces everywhere (while trying to identify and name the author). Everybody may contribute such a small article for inclusion.

It is best used together with Alexander Mai's Unix to OS/2 Porting FAQ¹. If his concise notes are too short for your purposes, some explanation may be found here.

The first chapter 'Instead of an Introduction' is, what a novice should read. Those who want to use imake should read the respective chapter, etc.

Bochum, in November 1999, Arnd Hanses

Ok, just let me begin, resuming and continuing an ongoing dialog in the net, thus introducing how to efficiently use the lists contributions and their archives (better insights always welcome):

¹Cf. the references below.

Part I

The Tutorial

Chapter 1

Instead of an Introduction

1.1 Why Are the Emx Development Tools Important for Writing Modern Portable Code ?

The number of new compilers for OS/2, which provide all (now standardized or still experimental) features of bleeding edge STL versions is very limited. 'Workframe' solutions fail, when a huge codebase with hundreds of megabytes of source is involved and a reasonable make process (i.e. based on Unix' 20 years of scientific and industrial experience with complex projects for heterogenous platforms and distributed networking, using quality software engineering strategies for code reliability) is required for any serious development. A prominent example is XFree86/OS2, where all make utilities and development tools except the GNU make utility and the GNU compiler `gcc` failed. Both are based on the `emx` libraries:

1.1.1 What's `emx`?

[excerpt from `emx 0.9d INTRODUCTION 21-Dec-1998`]

"Introduction

Welcome to `emx 0.9d`, an environment for creating 32-bit programs for OS/2 and DOS. You can use the GNU C compiler to compile programs for `emx`. The main design goal of `emx` is to simplify porting Unix software to OS/2 and DOS. Moreover, you can create 'native' OS/2 programs, including Presentation Manager applications.

The `emx` package includes:

- `emx.dll`, `emxio.dll`, `emxwrap.dll` and `emxlibc.dll` dynamic link libraries for OS/2
- `emx.exe` DOS extender for running 32-bit programs under DOS
- `emxfpemu`, W. Metzenthien's FPU (387) emulator

- `emxbind.exe` for creating `.exe` files which work both under OS/2 and DOS
- `emxomf.exe` for converting `.o` object files (`a.out`) to `.obj` object files (OMF). With `emxomf` and `LINK386` you can create OS/2 programs that don't need `emx.dll`
- `emximp.exe` for creating import libraries
- `emxexp.exe` for creating a list of export definitions from object files
- C header files and a complete C library
- a simple graphics library for 320x200 (256 colors) mode
- a library for screen output in windows
- header files and a library for using sockets of IBM TCP/IP for OS/2

Additionally, the following GNU programs are available compiled and with patches and sources:

- GCC 2.8.1, the GNU C compiler, supporting the C, C++, and Objective C languages
- GAS 2.6, the GNU assembler
- GDB 4.16, the GNU debugger
- `ld`, an ancient version of the GNU linker hacked for `emx`
- `strip`: a member of an ancient version of the GNU binary utilities
- `ar`, `nm`, `size`, `objdump`: a subset of the GNU binary utilities 2.6
- `gprof`, the BSD/GNU profiler
- `texinfo`, the GNU documentation system
- Patches for the GNU sources
- Patched source for `GCC`, `GAS`, `GDB`, `ld`, `ar`, `nm`, `size`, `strip`, `objdump`, `info`, `makeinfo`, `texindex`. You can compile all these programs with the files that come with `emx` (but you also need a make utility, such as `dmake`).

You can get the complete unpatched GNU sources by anonymous ftp from `prep.ai.mit.edu` and other archives such as `ftp.leo.org`.

Additionally, the following libraries are provided:

- some parts of the BSD C library
- the BSD data base library: `dbopen()`
- BSD curses library
- Henry Spencer's `regex` library
- The GNU and BSD `termcap` libraries [...]

After unpacking the `emx` runtime and development system packages, you'll find further information in the following files:

1.1. WHY ARE THE EMX DEVELOPMENT TOOLS IMPORTANT FOR WRITING MODERN PORTABLE CODE

- `\emx\doc\COPYING.EMX` emx & library license, license overview
- `\emx\doc\COPYING` GNU General Public License
- `\emx\doc\install.doc` Installation guide
- `\emx\doc\emxrt.doc` User's guide to the emx runtime package
- `\emx\doc\emxdev.doc` Application developer's guide
- `\emx\doc\emxlib.doc` C library reference
- `\emx\doc\faq.doc` Frequently asked questions (with answers)
- `\emx\doc\build.doc` Compiling the sources
- `\emx\doc\history.doc` Change log [...]

After unpacking the GNU and BSD packages, you'll find further information in the following files: [...]

- `\emx\doc\emxgnu.doc` GNU development tools information
- `\emx\doc\emxbsd.doc` BSD library information [...]

emx is available for anonymous ftp on

- `ftp.leo.org: /pub/comp/os/os2/leo/gnu/emx+gcc/`
- `ftp-os2.cdrom.com: /pub/os2/emx09d/`
- `ftp-os2.nmsu.edu: /pub/os2/dev/emx/v0.9d/ [...]`

An emx-related mailing list has been created. The address for people to request to be added to or removed from the list is:

`majordomo@iaehv.nl`

To subscribe, send a message containing

`subscribe emx`

to `majordomo@iaehv.nl`. [...]

This may be the right moment now to carefully read `emx/doc/INSTALL` file, and to test if you can successfully compile, run and debug the programs mentioned there.

Highly optimized and bleeding edge C++-code can be compiled with the most recent experimental GNU Pentium Compiler Group's `pgcc`, which also contains tools updates, interesting for assembler (MMX) programmers.

1.1.2 What's That Hype About "Code Reuse"?

Although not completely POSIX compliant, emx for OS/2 is nearly as close as can be for the pre-POSIX operating system OS/2. This makes it an ideal platform for CODE REUSE and OPEN PORTABILITY.

" \mathcal{X} - a portable network transparent window system", as the authors call it, enables the reuse of Unix software for OS/2 and in general gives you the chance to write extremely portable gui code, running on a PC as well as on a super-computer by simply recompiling it, provided you have a POSIX-like programming interface.

"Why is writing portable code (and therefore emx) so important?"

"As for non-existing API, I would recommend to stick to well-known things like POSIX, BSD networking, X11 and avoid system-specific calls like DosWrite() etc. completely. If you're doing really multiplatform project. Really 'standard' calls are the same everywhere and rarely require re-implementation. There's some oddity here and there (I heard Ultrix(?) does not have strdup())? but you can add it in transparent way, namely linking on Ultrix with your own implementation of strdup()."

(Dr. Sergey Ayukov, Sternberg Astronomical Institute Moscow, Russia <http://www.ayukov.com>, <http://crydee.sai.msu.ru/index-asv.html>)

Well, the tutorials motto is:

"Good programmers know what to write. Great ones know what to rewrite (and reuse)."

Paramount for reuse in the future is reusability through open standards (and - even more important - good comments ;-). Here the open Posix/BSD/MIT API's distinguish XFree/emx/gcc/pgcc from other compilers and make it the 'future compatible' development platform. For this reason it is outstanding and may well be recommended.

- "What do I need to know?"
Why? In short: Everything.
- "I don't have the necessary time?"
Then just start the emx compilation. Learn and debug by and by. Look at Alexander Mai's emx faq to get started.

Nevertheless we should try to explain what those 'X' files' are about (an informal glossary of uncommon terms and related hints) and provide some external references.

1.2 Where to Start?

"Heelpp, I'm stuck! Where to begin?"

(anonymous)

A good tutorial to ANSI C (cf. e.g. "An Introduction to C Programming", by Carsten Whimster

<http://www.edm2.com/0408/introc1.html>

is certainly not bad, but even for the beginner's purposes often insufficient. You will stumble over unknown concepts and interfaces, such as traditional 'K&R-style' 'BSD Unix' sources/sockets, the general preference to use exclusively the uncommon POSIX extensions, notwithstanding widespread adherence to the GNU language and C Library ("glibc") extensions. This tends to 'pollute' nearly every app coming from the UNIX world of operating systems. What you certainly would expect are Xlib (MIT X Standard) and Xt, Xaw, Xext, etc. library and toolkit extensions, as well as gui toolkits with rapidly changing popularity, namely 'qt', 'KDE', 'gnome', 'xforms', 'Motif/CDE' or 'lesstif', etc. We will pronounce now and here the well-meant advice:

"Use standard ANSI C (or C++) wherever possible!"

Only, in this moment this will probably help you less than nothing. Moreover, the only viable solution often requires low-level functions. Remember that 'EMX/GCC' as well as 'x11make' of XFree86/OS2 (or 'gmake') is generally a superset of the Unix standard 'cc', 'cxx' and 'make'. So, if you like to read a basic tutorial for beginners with a strictly didactical approach, I recommend as a good starting point:

"ANSI C for Programmers on UNIX Systems",
by T. P. Love, CUED Cambridge University Engineering Department)
<tpl@eng.cam.ac.uk>,

ftp://svr-ftp.eng.cam.ac.uk/pub/misc/love_C.ps.Z"

'Sunsite' mirrors distribute it in the 'tutorial' directory (they are the Unix World's first place to dig for source code, help and information).

In the following, to fill some gaps, I'll give a very short summary of porting-related concepts and specialties. For general information please refer to the online documentation that comes with recent Linux and FreeBSD distributions. It is also necessary that you have access to a good Unix/Linux tutorial, explaining those numerous concepts unfamiliar for OS/2, which cannot be covered here. For standards, try e.g.:

<http://standards.ieee.org/announcements/opengroup.html>

Before you start writing much maths-related code, check to see that it hasn't all been done before. Many maths routines, including routines that offer arbitrary precision are available by ftp from [netlib.att.com](ftp://netlib.att.com).

This tutorial will concentrate on emx-specific questions that recently were discussed on the respective lists. Further sources of information are indexed by Timur Tabi:

<http://www.edm2.com/links/index.html>

Some info for a better initial understanding of XFree86/OS2 can be found at:

http://hayai.freeshell.org/os2_xfree86.html

Chapter 2

Survive With Emx Tools (cc, make, gdb, emacs)

"Those who do not understand Unix are condemned to reinvent it, poorly."
(Henry Spencer)

2.1 Let's contrast Emx With FreeBSD.

The following are some excerpts from a terse introduction into how to behave in a FreeBSD Unix development environment ("A User's Guide to FreeBSD Programming Tools")

<http://www.de.freebsd.org/tutorials>.

This excerpt - with few extra comments - is meant to contrast emx and FreeBSD based Posix development (historic BSD Unix, besides SYSV, is the most prominent root of what later was distilled to the Posix interface), i.e. to demonstrate what is behind the well-known advice, emx gurus will give you:

"Simply proceed as if you were under Unix!"

I'm sure, you'll soon discover how little a programmer who is writing portable ('posixified') applications has to re-learn or re-do.

2.1.1 Compiling With cc

"This section deals only with the GNU compiler for C and C++, since that comes with the base FreeBSD [and the emx] system. It can be invoked by either cc [write a cc.cmd file] or gcc. ... Once you've written your masterpiece, the next step is to convert it into something that will (hopefully!) run on FreeBSD [as well as on OS/2 after compiling your portable Posix source with emx gcc]. This usually involves several steps, each of which is done by a separate program:

1. Pre-process your source code to remove comments and do other tricks like expanding macros in C.
 - (a) Check the syntax of your code to see if you have obeyed the rules of the language. If you have not, it will complain! [Cf. the different warning options: Use at least `-Wall`.]
 - (b) Convert the source code into assembly language - this is very close to machine code, but still understandable by humans. Allegedly.
 - (c) Convert the assembly language into machine code - yep, we are talking bits and bytes, ones and zeros here.
 - (d) Check that you have used things like functions and global variables in a consistent way. For example, if you have called a non-existent function, it will complain. [Something like *'Unresolved reference foo in bar'*: Try to find out where you can steal the library, which contains this function or variable. Or maybe you forgot to specify `-foolib` on the command line?]
 - (e) If you are trying to produce an executable from several source code files, work out how to fit them all together. [This is the fixup of unresolved symbols, it is done by the linker `ld` or `link386` (if you specified `-Zomf`), respectively.]
 - (f) Work out how to produce something that the system's run-time loader will be able to load into memory and run. [`emxbind` or `link386`]
 - (g) Finally, write the executable on the file system.

The word compiling is often used to refer to just steps 1 to 4—the others are referred to as linking. Sometimes step 1 is referred to as pre-processing and steps 3-4 as assembling. Fortunately, almost all this detail is hidden from you, as `cc` is a front end that manages calling all these programs with the right arguments for you; simply typing

```
% cc foobar.c
```

will cause `foobar.c` to be compiled by all the steps above. If you have more than one file to compile, just do something like

```
% cc foo.c bar.c
```

Note that the syntax checking is just that - checking the syntax. It will not check for any logical mistakes you may have made, like putting the program into an infinite loop, or using a bubble sort when you meant to use a binary sort.

There are lots and lots of options for `cc`, which are all in the man page. Here are a few of the most important ones, with examples of how to use them.

```
-o filename
```

The output name of the file [containing a trailing `.exe` or `-Zexe` switch]. If you do not use this option, `cc` will produce an executable called `a.out`.

```
% cc -c foobar.c
```

Just compile the file, do not link it. Useful for toy programs where you just want to check the syntax, or if you are using a Makefile. This will produce an object file (not an executable) called `foobar.o`. This can be linked together with other object files into an executable.

```
% cc -g foobar.c
```

Create a debug version of the executable. This makes the compiler put information into the executable about which line of which source file corresponds to which function call. A debugger can use this information to show the source code as you step through the program, which is very useful; the disadvantage is that all this extra information makes the program much bigger.

```
% cc -O -o foobar foobar.c
```

Create an optimised version of the executable. The compiler performs various clevertricks to try and produce an executable that runs faster than normal. You can add a number after the `-O` to specify a higher level of optimisation, but this often exposes bugs in the compiler's optimiser. ... Optimisation is usually only turned on when compiling a release version. ...

The following three flags will force `cc` to check that your code complies to the relevant international standard, often referred to as the ANSI standard, though strictly speaking it is an ISO standard.

```
-Wall
```

Enable all the warnings which the authors of `cc` believe are worthwhile. Despite the name, it will not enable all the warnings `cc` is capable of.

```
-ansi
```

Turn off most, but not all, of the non-ANSI/C features provided by `cc`. Despite the name, it does not guarantee strictly that your code will comply to the standard.

```
-pedantic
```

Turn off all `cc`'s non-ANSI/C features.

Without these flags, `cc` will allow you to use some of its non-standard extensions to the standard. Some of these are very useful, but will not work with other compilers - in fact, one of the main aims of the standard is to allow people to write code that will work with any compiler on any system. This is known as portable code. Generally, you should try to make your code as portable as possible, as otherwise you may have to completely re-write the program later to get it to work somewhere else - and who knows what you may be using in a few years time?

```
% cc -Wall -ansi -pedantic -o foobar.exe foobar.c
```

This will produce an executable foobar after checking foobar.c for standard compliance.

```
% cc -o foobar foobar.c -lm
```

Specify a function library to be used during when linking.

The most common example of this is when compiling a program that uses some of the mathematical functions in C. Unlike most other platforms [emx included], these are in a separate library from the standard C one and you have to tell the compiler to add it.

The rule is that if the library is called `libsomething.a`, you give `cc` the argument `-lsomething`. For example, the math library is `libm.a`, so you give `cc` the argument `-lm`. A common "gotcha" with the math library is that it has to be the last library on the command line. [Emx does not cut off the leading 'lib' part of library names and `m.a` or `m.lib` is just a dummy library which can be used for your own common extension modules.]

If you are compiling C++ code, you need to add `-lg++`, or `-lstdc++` ... to the command line argument to link the C++ library functions. ...

Each of these will both produce an executable foobar from the C++ source file foobar.cc. Note that, on Unix systems, C++ source files traditionally end in `.C`, `.cxx` or `.cc`, rather than the MS-DOS style `.cpp` (which was already used for something else). `gcc` used to rely on this to work out what kind of compiler to use on the source file; however, this restriction no longer applies, so you may now call your C++ files `.cpp` with impunity!

2.1.2 Common cc Queries and Problems

Q: I am trying to write a program which uses the `sin()` function and I get an error like this. What does it mean?

```
/var/tmp/cc0143941.o: Undefined symbol '_sin' referenced from text segment
```

A: When using mathematical functions like `sin()`, you have to [you need not, since emx does it for you] tell `cc` to link in the math library. [Nevertheless the error occurs if you are missing something else.]

Q: All right, I wrote this simple program to practice using `-lm`. All it does is raise 2.1 to the power of 6.

```
#include <stdio.h>

int main()
{
```

```
float f = pow(2.1, 6);
    printf("2.1 ^ 6 = %f", f);
    return 0;
}
```

and I compiled it as:

```
% cc temp.c -lm
```

like you said I should, but I get this when I run it:

```
% ./a.out
2.1 ^ 6 = 1023.000000
```

This is not the right answer! What is going on?

A: When the compiler sees you call a function, it checks if it has already seen a prototype for it. If it has not, it assumes the function returns an int, which is definitely not what you want here.

Q: So how do I fix this?

A: The prototypes for the mathematical functions are in `math.h`. If you include this file, the compiler will be able to find the prototype and it will stop doing strange things to your calculation!

```
#include <math.h>
#include <stdio.h>

int main() {
    ...
}
```

After recompiling it as you did before, run it:

```
% ./a.out
2.1 ^ 6 = 85.766121
```

If you are using any of the mathematical functions, always include `math.h` and remember to link in the math library. [This problem can occur with any missing header. Let the compiler warn you: Always use the *-Wall* flag!]

Q: I compiled my program and it seemed to run all right at first, then there was an error and it said something about core dumped. What does that mean?

A: The name core dump dates back to the very early days of Unix, when the machines used core memory for storing data. Basically, if the program failed under certain conditions, the system would write the contents of core memory to disk in a file called core, which the programmer could then pore over to find out what went wrong.

Q: Fascinating stuff, but what I am supposed to do now?

A: Use gdb to analyse the core (see Section 5).

Q: When my program dumped core, it said something about a segmentation fault. What's that?

A: This basically means that your program tried to perform some sort of illegal operation on memory; Unix is designed to protect the operating system and other programs from rogue programs.

Common causes for this are [Read this carefully, well meant advice]:

Trying to write to a NULL pointer, eg

```
char *foo = NULL;
strcpy(foo, "bang!");
```

Using a pointer that hasn't been initialised, eg

```
char *foo;
strcpy(foo, "bang!");
```

Trying to access past the end of an array, eg

```
int bar[20];
bar[27] = 6;
```

Trying to store something in read-only memory, e.g.

```
char *foo = "My string";
strcpy(foo, "bang!");
```

Unix compilers often put string literals like "My string" into read-only areas of memory. [Declare constants always explicitly as such:

```
const char *foo = "My string"; /* constant! */
```

Now the compiler will warn you. Remember: Let the compiler warn you: Always use the *-Wall* flag!]

Doing naughty things with malloc() and free(), eg

```
char bar[80];
free(bar);
```

or

```
char *foo = malloc(27);
free(foo);
free(foo);
```

Making one of these mistakes will not always lead to an error, but they are always bad practice. Some systems and compilers are more tolerant than others, which is why programs that ran well on one system can crash when you try them on another.

2.1.3 Make

2.1.3.1 What's make?

It reads in a file, called a makefile, that tells it how different files depend on each other, and works out which files need to be re-compiled and which ones don't. For example, a rule could say something like "if fromboz.o is older than fromboz.c, that means someone must have changed fromboz.c, so it needs to be re-compiled." The makefile also has rules telling make how to re-compile the source file, making it a much more powerful tool.

Makefiles are typically kept in the same directory as the source they apply to, and can be called makefile, Makefile or MAKEFILE. Most programmers use the name Makefile, as this puts it near the top of a directory listing, where it can easily be seen.

2.1.3.2 Example of using make

Here's a very simple make file:

```
foo: foo.c
    cc -o foo foo.c
```

It consists of two lines, a dependency line and a creation line. The dependency line here consists of the name of the program (known as the TARGET), followed by a colon, then whitespace, then the name of the source file. When make reads this line, it looks to see if `foo` exists; if it exists, it compares the time `foo` was last modified to the time `foo.c` was last modified. If `foo` does not exist, or is older than `foo.c`, it then looks at the creation line to find out what to do. In other words, this is the rule for working out when `foo.c` needs to be re-compiled.

The creation line starts with a tab (press the tab key) and then the command you would type to create `foo` if you were doing it at a command prompt. If `foo` is out of date, or does not exist, make then executes this command to create it. In other words, this is the rule which tells make how to re-compile `foo.c`.

So, when you type make, it will make sure that `foo` is up to date with respect to your latest changes to `foo.c`. This principle can be extended to Makefiles with hundreds of targets—in fact, on FreeBSD, it is possible to compile the entire operating system just by typing `make world` in the appropriate directory! Another useful property of makefiles is that the targets don't have to be programs. For instance, we could have a make file that looks like this:

```
foo: foo.c
    cc -o foo.exe foo.c

install:
    cp foo /home/me
```

We can tell make which target we want to make by typing:

```
% make target
```

make will then only look at that target and ignore any others. For example, if we type `make foo` with the makefile above, make will ignore the `install` target.

If we just type `make` on its own, make will always look at the first target and then stop without looking at any others. So if we typed `make here`, it will just go to the `foo` target, re-compile `foo` if necessary, and then stop without going on to the `install` target.

Notice that the `install` target doesn't actually depend on anything! This means that the command on the following line is always executed when we try to make that target by typing `make install`. In this case, it will copy `foo` into the user's home directory. This is often used by application makefiles, so that the application can be installed in the correct directory when it has been correctly compiled.

This is a slightly confusing subject to try and explain. If you don't quite understand how make works, the best thing to do is to write a simple program like "hello world" and a make file like the one above and experiment. Then progress to using more than one source file, or having the source file include a header file. The `touch` command is very useful here—it changes the date on a file without you having to edit it. ...

2.1.3.3 More advanced uses of make

Make is a very powerful tool, and can do much more than the simple example above shows. Unfortunately, there are several different versions of make, and they all differ considerably. The best way to learn what they can do is probably to read the documentation—hopefully this introduction will have given you a base from which you can do this.

The version of make that comes with FreeBSD is the Berkeley make ... Many applications in the ports use GNU make, which has a very good set of "info" pages. If you have installed any of these ports, GNU make will automatically have been installed as `gmake`. It's also available as a port and package in its own right.

To view the info pages for GNU make, you will have to edit the `dir` file in the `/usr/local/info` directory to add an entry for it. This involves adding a line like

```
* Make: (make).                The GNU Make utility.
```

to the file. Once you have done this, you can type `info` and then select `make` from the menu (or in Emacs, do `C-h i`).

2.2 Debugging

2.2.1 The Debugger

The debugger that comes with FreeBSD [and emx] is called `gdb` (GNU debugger). You start it up by typing

```
% gdb progname
```

although most people prefer to run it inside Emacs. You can do this by:

```
M-x gdb RET progname RET
```

Using a debugger allows you to run the program under more controlled circumstances. Typically, you can step through the program a line at a time, inspect the value of variables, change them, tell the debugger to run up to a certain point and then stop, and so on. You can even attach to a program that's already running, or load a core file to investigate why the program crashed. ... `gdb` has quite good on-line help, as well as a set of info pages, so this section will concentrate on a few of the basic commands. Finally, if you find its text-based command-prompt style off-putting, there's a graphical

front-end for it `xxgdb` in the ports collection. [Emx calls it `pmgdb`.] ...

2.2.2 Running a program in the debugger

You'll need to have compiled the program with the `-g` option to get the most out of using `gdb`. It will work without, but you'll only see the name of the function you're in, instead of the source code. If you see a line like:

```
... (no debugging symbols found) ...
```

when `gdb` starts up, you'll know that the program wasn't compiled with the `-g` option.

At the `gdb` prompt, type `break main`. This will tell the debugger to skip over the preliminary set-up code in the program and start at the beginning of your code. Now type `run` to start the program - it will start at the beginning of the set-up code and then get stopped by the debugger when it calls `main()`. (If you've ever wondered where `main()` gets called from, now you know!).

You can now step through the program, a line at a time, by pressing `n`. If you get to a function call, you can step into it by pressing `s`. Once you're in a function call, you can return from stepping into a function call by pressing `f`. You can also use `up` and `down` to take a quick look at the caller.

Here's a simple example of how to spot a mistake in a program with `gdb`. This is our program (with a deliberate mistake):

```

#include <stdio.h>

int bazz(int anint);

main() {
    int i;

    printf("This is my program\n");
    bazz(i);
    return 0;
}

int bazz(int anint) {
    printf("You gave me %d\n", anint);
    return anint;
}

```

This program sets `i` to be 5 and passes it to a function `bazz()` which prints out the number we gave it.

When we compile and run the program we get

```

% cc -g -o temp temp.c
% ./temp
This is my program
anint = 4231

```

That wasn't what we expected! Time to see what's going on!

```

% gdb temp
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc.
(gdb) break main                               Skip the set-up code
Breakpoint 1 at 0x160f: file temp.c, line 9. breakpoint at main()
(gdb) run                                       Run as far as main()
Starting program: temp                          Program starts running

Breakpoint 1, main () at temp.c:9             gdb stops at main()
(gdb) n                                       Go to next line
This is my program                             Program prints out
(gdb) s                                       step into bazz()
bazz (anint=4231) at temp.c:17                gdb displays stack frame
(gdb)

```

Hang on a minute! How did `anint` get to be 4231? Didn't we set it to be 5 in `main()`? Let's move up to `main()` and have a look.

```

(gdb) up                                Move up call stack
#1 0x1625 in main () at temp.c:11 gdb displays stack frame
(gdb) p i                                Show us the value of i
$1 = 4231                                gdb displays 4231

```

Oh dear! Looking at the code, we forgot to initialise `i`. We meant to put

```

...
main() {
    int i;

    i = 5;
    printf("This is my program\n");
...

```

but we left the `i=5;` line out. As we didn't initialise `i`, it had whatever number happened to be

in that area of memory when the program ran, which in this case happened to be 4231.

Note: `gdb` displays the stack frame every time we go into or out of a function, even if we're using `up` and `down` to move around the call stack. This shows the name of the function and the values of its arguments, which helps us keep track of where we are and what's going on. (The stack is a storage area where the program stores information about the arguments passed to functions and where to go when it returns from a function call).

2.2.3 Examining a core file

A core file is basically a file which contains the complete state of the process when it crashed. In "the good old days", programmers had to print out hex listings of core files and sweat over machine code manuals, but now life is a bit easier. Incidentally, under FreeBSD and other 4.4BSD systems, a core file is called `progname.core` instead of just `core`, to make it clearer which program a core file belongs to. [`emx` calls it just `core`.]

To examine a core file, start up `gdb` in the usual way. Instead of typing `break` or `run`, type

```
(gdb) core progname.core [emx: gdb progname core]
```

If you're not in the same directory as the core file, you'll have to do `dir /path/to/core/file` first.

You should see something like this:

```

% gdb a.out
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc.
(gdb) core a.out.core
Core was generated by 'a.out'.
Program terminated with signal 11, Segmentation fault.
Cannot access memory at address 0x7020796d.
#0  0x164a in bazz (anint=0x5) at temp.c:17
(gdb)

```

In this case, the program was called `a.out`, so the core file is called `a.out.core`. We can see that the program crashed due to trying to access an area in memory that was not available to it in a function called `bazz`.

Sometimes it's useful to be able to see how a function was called, as the problem could have occurred a long way up the call stack in a complex program. The `bt` command causes `gdb` to print out a back-trace of the call stack:

```

(gdb) bt
#0  0x164a in bazz (anint=0x5) at temp.c:17
#1  0xefbfd888 in end()
#2  0x162c in main() at temp.c:11
(gdb)

```

The `end()` function is called when a program crashes; in this case, the `bazz()` function was called from `main()`.

2.2.4 Attaching to a running program

One of the neatest features about `gdb` is that it can attach to a program that's already running. Of course, that assumes you have sufficient permissions to do so. A common problem is when you are stepping through a program that forks, and you want to trace the child, but the debugger will only let you trace the parent.

What you do is start up another `gdb`, use `ps` to find the process ID for the child, and do

```
(gdb) attach pid
```

in `gdb`, and then debug as usual.

"That's all very well," you're probably thinking, "but by the time I've done that, the child process will be over the hill and far away". Fear not, gentle reader, here's how to do it (courtesy of the `gdb` info pages):

```

...
if ((pid = fork()) < 0)    /* _Always_ check this */
    error();
else if (pid == 0) {      /* child */
    int PauseMode = 1;

    while (PauseMode)
        sleep(10); /* Wait until someone attaches to us */
    ...
} else {                  /* parent */
    ...

```

Now all you have to do is attach to the child, set `PauseMode` to 0, and wait for the `sleep()` call to return!

Using Emacs as a Development Environment

Emacs

[...] Emacs is basically a highly customisable editor—indeed, it has been customised to the point where it’s more like an operating system than an editor! Many developers and sysadmins do in fact spend practically all their time working inside Emacs, leaving it only to log out. [I recommend also to test `elvis`, a `vi` clone.] It’s impossible even to summarise everything Emacs can do here, but here are some of the features of interest to developers:

- Very powerful editor, allowing search-and-replace on both strings and regular expressions (patterns), jumping to start/end of block expression, etc, etc.
- Pull-down menus and online help.
- Language-dependent syntax highlighting and indentation.
- Completely customisable.
- You can compile and debug programs within Emacs.
- On a compilation error, you can jump to the offending line of source code.
- Friendly frontend to the `info` program used for reading GNU hypertext documentation, including the documentation on Emacs itself.
- Friendly front-end to `gdb`, allowing you to look at the source code as you step through your program.
- You can read Usenet news and mail while your program is compiling.
- And doubtless many more that I’ve overlooked.

Emacs can be installed on FreeBSD [and on `emx`] using the Emacs port.

Once it’s installed, start it up and do `C-h t` to read an Emacs tutorial—that means hold down the control key, press `h`, let go of the control key, and then

press `t`. (Alternatively, you can use the mouse to select Emacs Tutorial from the Help menu).

Although Emacs does have menus, it's well worth learning the key bindings, as it's much quicker when you're editing something to press a couple of keys than to try and find the mouse and then click on the right place. And, when you're talking to seasoned Emacs users, you'll find they often casually throw around expressions like "`M-x replace-s RET foo RET bar RET`" so it's useful to know what they mean. And in any case, Emacs has far too many useful functions for them to all fit on the menu bars.

Fortunately, it's quite easy to pick up the key-bindings, as they're displayed next to the menu item. My advice is to use the menu item for, say, opening a file until you understand how it works and feel confident with it, then try doing `C-x C-f`. When you're happy with that, move on to another menu command.

If you can't remember what a particular combination of keys does, select **Describe Key** from the Help menu and type it in—Emacs will tell you what it does. You can also use the **Command Apropos** menu item to find out all the commands which contain a particular word in them, with the key binding next to it.

By the way, the expression above means hold down the Meta key, press `x`, release the Meta key, type `replace-s` (short for replace-string—another feature of Emacs is that you can abbreviate commands), press the return key, type `foo` (the string you want replaced), press the return key, type `bar` (the string you want to replace `foo` with) and press return again. Emacs will then do the search-and-replace operation you've just requested. If you're wondering what on earth the Meta key is, it's a special key that many Unix workstations have. Unfortunately, PC's don't have one, so it's usually the alt key (or if you're unlucky, the escape key).

Oh, and to get out of Emacs, do `C-x C-c` (that means hold down the control key, press `x`, press `c` and release the control key). If you have any unsaved files open, Emacs will ask you if you want to save them. (Ignore the bit in the documentation where it says `C-z` is the usual way to leave Emacs—that leaves Emacs hanging around in the background, and is only really useful if you're on a system which doesn't have virtual terminals).

2.3 Configuring Emacs

Emacs does many wonderful things; some of them are built in, some of them need to be configured.

Instead of using a proprietary macro language for configuration, Emacs uses a version of Lisp specially adapted for editors, known as Emacs Lisp. This can be quite useful if you want to go on and learn something like Common Lisp, as it's considerably smaller than Common Lisp (although still quite big!).

The best way to learn Emacs Lisp is to download the Emacs Tutorial. However, there's no need to actually know any Lisp to get started with configuring Emacs, as I've included a sample `.emacs` file, which should be enough to get you started. Just copy it into your home directory and restart Emacs if it's already running;

it will read the commands from the file and (hopefully) give you a useful basic setup.

2.3.1 A sample .emacs file

Unfortunately, there's far too much here to explain it in detail; however there are one or two points worth mentioning.

- Everything beginning with a `;` is a comment and is ignored by Emacs.
- In the first line, the `--Emacs-Lisp--` is so that we can edit the `.emacs` file itself within Emacs and get all the fancy features for editing Emacs Lisp. Emacs usually tries to guess this based on the filename, and may not get it right for `.emacs`.

The `tab` key is bound to an indentation function in some modes, so when you press the `tab` key, it will indent the current line of code. If you want to put a `tab` character in whatever you're writing, hold the `control` key down while you're pressing the `tab` key.

This file supports syntax highlighting for C, C++, Perl, Lisp and Scheme, by guessing the language from the filename.

Emacs already has a pre-defined function called `next-error`. In a compilation output window, this allows you to move from one compilation error to the next by doing `M-n`; we define a complementary function, `previous-error`, that allows you to go to a previous error by doing `M-p`. The nicest feature of all is that `C-c C-c` will open up the source file in which the error occurred and jump to the appropriate line.

We enable Emacs's ability to act as a server, so that if you're doing something outside Emacs and you want to edit a file, you can just type in

```
% emacsclient filename
```

and then you can edit the file in your Emacs! Many Emacs users set their `EDITOR` environment to `emacsclient` so this happens every time they need to edit a file.

2.3.2 Example A sample .emacs file

```
;; --Emacs-Lisp--

;; This file is designed to be re-evald; use the variable first-time
;; to avoid any problems with this.
(defvar first-time t
  "Flag signifying this is the first time that .emacs has been evald")

;; Meta
```

```

(global-set-key "\M- " 'set-mark-command)
(global-set-key "\M-\C-h" 'backward-kill-word)
(global-set-key "\M-\C-r" 'query-replace)
(global-set-key "\M-r" 'replace-string)
(global-set-key "\M-g" 'goto-line)
(global-set-key "\M-h" 'help-command)

;; Function keys
(global-set-key [f1] 'manual-entry)
(global-set-key [f2] 'info)
(global-set-key [f3] 'repeat-complex-command)
(global-set-key [f4] 'advertised-undo)
(global-set-key [f5] 'eval-current-buffer)
(global-set-key [f6] 'buffer-menu)
(global-set-key [f7] 'other-window)
(global-set-key [f8] 'find-file)
(global-set-key [f9] 'save-buffer)
(global-set-key [f10] 'next-error)
(global-set-key [f11] 'compile)
(global-set-key [f12] 'grep)
(global-set-key [C-f1] 'compile)
(global-set-key [C-f2] 'grep)
(global-set-key [C-f3] 'next-error)
(global-set-key [C-f4] 'previous-error)
(global-set-key [C-f5] 'display-faces)
(global-set-key [C-f8] 'dired)
(global-set-key [C-f10] 'kill-compilation)

;; Keypad bindings
(global-set-key [up] "\C-p")
(global-set-key [down] "\C-n")
(global-set-key [left] "\C-b")
(global-set-key [right] "\C-f")
(global-set-key [home] "\C-a")
(global-set-key [end] "\C-e")
(global-set-key [prior] "\M-v")
(global-set-key [next] "\C-v")
(global-set-key [C-up] "\M-\C-b")
(global-set-key [C-down] "\M-\C-f")
(global-set-key [C-left] "\M-b")
(global-set-key [C-right] "\M-f")
(global-set-key [C-home] "\M-<")
(global-set-key [C-end] "\M->")
(global-set-key [C-prior] "\M-<")
(global-set-key [C-next] "\M->")

;; Mouse
(global-set-key [mouse-3] 'imenu)

;; Misc

```

```

(global-set-key [C-tab] "\C-q\t") ; Control tab quotes a tab.
(setq backup-by-copying-when-mismatch t)

;; Treat 'y' or <CR> as yes, 'n' as no.
(fset 'yes-or-no-p 'y-or-n-p)
  (define-key query-replace-map [return] 'act)
  (define-key query-replace-map [?\C-m] 'act)

;; Load packages
(require 'desktop)
(require 'tar-mode)

;; Pretty diff mode
(autoload 'ediff-buffers "ediff" "Intelligent Emacs interface to diff" t)
(autoload 'ediff-files "ediff" "Intelligent Emacs interface to diff" t)
(autoload 'ediff-files-remote "ediff"
  "Intelligent Emacs interface to diff")

(if first-time
  (setq auto-mode-alist
    (append '(("\\.cpp$" . c++-mode)
              ("\\.hpp$" . c++-mode)
              ("\\.lsp$" . lisp-mode)
              ("\\.scm$" . scheme-mode)
              ("\\.pl$" . perl-mode)
              ) auto-mode-alist)))

;; Auto font lock mode
(defvar font-lock-auto-mode-list
  (list 'c-mode 'c++-mode 'c++-c-mode 'emacs-lisp-mode 'lisp-mode 'perl-mode 'scheme-mode)
  "List of modes to always start in font-lock-mode")

(defvar font-lock-mode-keyword-alist
  '((c++-c-mode . c-font-lock-keywords)
    (perl-mode . perl-font-lock-keywords))
  "Associations between modes and keywords")

(defun font-lock-auto-mode-select ()
  "Automatically select font-lock-mode if the current major mode is
in font-lock-auto-mode-list"
  (if (memq major-mode font-lock-auto-mode-list)
      (progn
        (font-lock-mode t))
      )
  )

(global-set-key [M-f1] 'font-lock-fontify-buffer)

;; New dabbrev stuff
(require 'new-dabbrev)

```

```

(setq dabbrev-always-check-other-buffers t)
(setq dabbrev-abbrev-char-regexp "\\sw\\|\\s_")
(add-hook 'emacs-lisp-mode-hook
  '(lambda ()
    (set (make-local-variable 'dabbrev-case-fold-search) nil)
    (set (make-local-variable 'dabbrev-case-replace) nil)))
(add-hook 'c-mode-hook
  '(lambda ()
    (set (make-local-variable 'dabbrev-case-fold-search) nil)
    (set (make-local-variable 'dabbrev-case-replace) nil)))
(add-hook 'text-mode-hook
  '(lambda ()
    (set (make-local-variable 'dabbrev-case-fold-search) t)
    (set (make-local-variable 'dabbrev-case-replace) t)))

;; C++ and C mode...
(defun my-c++-mode-hook ()
  (setq tab-width 4)
  (define-key c++-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key c++-mode-map "\C-ce" 'c-comment-edit)
  (setq c++-auto-hungry-initial-state 'none)
  (setq c++-delete-function 'backward-delete-char)
  (setq c++-tab-always-indent t)
  (setq c-indent-level 4)
  (setq c-continued-statement-offset 4)
  (setq c++-empty-arglist-indent 4))

(defun my-c-mode-hook ()
  (setq tab-width 4)
  (define-key c-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key c-mode-map "\C-ce" 'c-comment-edit)
  (setq c-auto-hungry-initial-state 'none)
  (setq c-delete-function 'backward-delete-char)
  (setq c-tab-always-indent t)
  ;; BSD-ish indentation style
  (setq c-indent-level 4)
  (setq c-continued-statement-offset 4)
  (setq c-brace-offset -4)
  (setq c-argdecl-indent 0)
  (setq c-label-offset -4))

;; Perl mode
(defun my-perl-mode-hook ()
  (setq tab-width 4)
  (define-key c++-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (setq perl-indent-level 4)
  (setq perl-continued-statement-offset 4))

;; Scheme mode...
(defun my-scheme-mode-hook ()

```

```

      (define-key scheme-mode-map "\C-m" 'reindent-then-newline-and-indent))

;; Emacs-Lisp mode...
(defun my-lisp-mode-hook ()
  (define-key lisp-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key lisp-mode-map "\C-i" 'lisp-indent-line)
  (define-key lisp-mode-map "\C-j" 'eval-print-last-sexp))

;; Add all of the hooks...
(add-hook 'c++-mode-hook 'my-c++-mode-hook)
(add-hook 'c-mode-hook 'my-c-mode-hook)
(add-hook 'scheme-mode-hook 'my-scheme-mode-hook)
(add-hook 'emacs-lisp-mode-hook 'my-lisp-mode-hook)
(add-hook 'lisp-mode-hook 'my-lisp-mode-hook)
(add-hook 'perl-mode-hook 'my-perl-mode-hook)

;; Complement to next-error
(defun previous-error (n)
  "Visit previous compilation error message and corresponding source code."
  (interactive "p")
  (next-error (- n)))

;; Misc...
(transient-mark-mode 1)
(setq mark-even-if-inactive t)
(setq visible-bell nil)
(setq next-line-add-newlines nil)
(setq compile-command "make")
(setq suggest-key-bindings nil)
(put 'eval-expression 'disabled nil)
(put 'narrow-to-region 'disabled nil)
(put 'set-goal-column 'disabled nil)

;; Elisp archive searching
(autoload 'format-lisp-code-directory "lispdir" nil t)
(autoload 'lisp-dir-afropos "lispdir" nil t)
(autoload 'lisp-dir-retrieve "lispdir" nil t)
(autoload 'lisp-dir-verify "lispdir" nil t)

;; Font lock mode
(defun my-make-face (face colour &optional bold)
  "Create a face from a colour and optionally make it bold"
  (make-face face)
  (copy-face 'default face)
  (set-face-foreground face colour)
  (if bold (make-face-bold face))
  )

(if (eq window-system 'x)
    (progn

```

```

(my-make-face 'blue "blue")
(my-make-face 'red "red")
(my-make-face 'green "dark green")
(setq font-lock-comment-face 'blue)
(setq font-lock-string-face 'bold)
(setq font-lock-type-face 'bold)
(setq font-lock-keyword-face 'bold)
(setq font-lock-function-name-face 'red)
(setq font-lock-doc-string-face 'green)
(add-hook 'find-file-hooks 'font-lock-auto-mode-select)

(setq baud-rate 1000000)
(global-set-key "\C-cmm" 'menu-bar-mode)
(global-set-key "\C-cms" 'scroll-bar-mode)
(global-set-key [backspace] 'backward-delete-char)
; (global-set-key [delete] 'delete-char)
(standard-display-european t)
(load-library "iso-transl"))

;; X11 or PC using direct screen writes
(if window-system
  (progn
    (global-set-key [M-f1] 'hilit-repaint-command)
    (global-set-key [M-f2] [?\C-u M-f1])
    (setq hilit-mode-enable-list
      '(not text-mode c-mode c++-mode emacs-lisp-mode lisp-mode
        scheme-mode)
      hilit-auto-highlight nil
      hilit-auto-rehighlight 'visible
      hilit-inhibit-hooks nil
      hilit-inhibit-rebinding t)
    (require 'hilit19)
    (require 'paren))
  (setq baud-rate 2400) ; For slow serial connections
)

;; TTY type terminal
(if (and (not window-system)
  (not (equal system-type 'ms-dos)))
  (progn
    (if first-time
      (progn
        (keyboard-translate ?\C-h ?\C-?)
        (keyboard-translate ?\C-? ?\C-h))))))

;; Under UNIX
(if (not (equal system-type 'ms-dos))
  (progn
    (if first-time
      (server-start))))

```

```

;; Add any face changes here
(add-hook 'term-setup-hook 'my-term-setup-hook)
(defun my-term-setup-hook ()
  (if (eq window-system 'pc)
      (progn
        ;; (set-face-background 'default "red")
      )))

;; Restore the "desktop" - do this as late as possible
(if first-time
    (progn
      (desktop-load-default)
      (desktop-read)))

;; Indicate that this file has been read at least once
(setq first-time nil)

;; No need to debug anything now
(setq debug-on-error nil)

;; All done
(message "All done, %s%s" (user-login-name) ".")

```

2.3.3 Extending the Range of Languages Emacs Understands

Now, this is all very well if you only want to program in the languages already catered for in the `.emacs` file (C, C++, Perl, Lisp and Scheme), but what happens if a new language called "whizbang" comes out, full of exciting features?

The first thing to do is find out if whizbang comes with any files that tell Emacs about the language. These usually end in `.el`, short for "Emacs Lisp". For example, if whizbang is a FreeBSD port, we can locate these files by doing

```
% find /usr/ports/lang/whizbang -name "*.el" -print
```

and install them by copying them into the Emacs site Lisp directory. On FreeBSD 2.1.0-RELEASE, this is `/usr/local/share/emacs/site-lisp`. [Depends on the `emx` port of your emacs.]

So for example, if the output from the `find` command was

```
/usr/ports/lang/whizbang/work/misc/whizbang.el
```

we would do

```
# cp /usr/ports/lang/whizbang/work/misc/whizbang.el /usr/local/share/emacs/site-lisp
```

Next, we need to decide what extension whizbang source files have. Let's say for the sake of argument that they all end in `.wiz`. We need to add an entry to our `.emacs` file to make sure Emacs will be able to use the information in `whizbang.el`.

Find the `auto-mode-alist` entry in `.emacs` and add a line for whizbang, such as:

```
...
("\\.lisp$" . lisp-mode)
("\\.wiz$" . whizbang-mode)
("\\.scm$" . scheme-mode)
...
```

This means that Emacs will automatically go into whizbang-mode when you edit a file

ending in `.wiz`.

Just below this, you'll find the `font-lock-auto-mode-list` entry. Add whizbang-mode to it like so:

```
;; Auto font lock mode
(defvar font-lock-auto-mode-list
  (list 'c-mode 'c++-mode 'c++-c-mode 'emacs-lisp-mode 'whizbang-mode 'lisp-
        "List of modes to always start in font-lock-mode"))
```

This means that Emacs will always enable `font-lock-mode` (ie syntax highlighting) when editing a `.wiz` file.

And that's all that's needed. If there's anything else you want done automatically when you open up a `.wiz` file, you can add a whizbang-mode hook (see `my-scheme-mode-hook` for a simple example that adds auto-indent).

2.4 Further Reading

- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (2nd Edition), Prentice-Hall 1988, ISBN 0-13-110362-8
- J.F. Korsh and L.J. Garrett, *Data Structures, Algorithms, and Program Style using C*, PSW-Kent 1988
- Samuel P. Harbison and Guy L. Steele, *C: A Reference Manual*, Addison-Wesley 1987
- Andrew Koenig, *C Traps and Pitfalls*, Addison-Wesley 1989
- Mark Horton, *Portable C Software*, Prentice-Hall 1990
- William H. Press and Brian P. Flannery and Saul A. Teukolsky and William T. Vetterling, *NUMERICAL RECIPES in C: The Art of Scientific Computing*, Cambridge University Press 1988

- Brian W. Kernighan and Rob Pike, *The Unix Programming Environment*, Prentice-Hall 1984, ISBN 0-13-937681-X
- W. Richard Stevens, *Advanced Programming in the Unix Environment*, Addison-Wesley 1992, ISBN 0-201-56317-7
- W. Richard Stevens, *Unix Network Programming*, Prentice-Hall 1990, ISBN 0-13-949876-1
- Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley 1991, ISBN 0-201-53992-6
- Winston and Horn, *Lisp* (3rd Edition), Addison-Wesley 1989, ISBN 0-201-08319-1

Part II

Special Problems

Chapter 3

Frequently Used Concepts

*"Too much \mathcal{X} around, I'm confused!"
(A novice to \mathcal{X})*

3.1 What's 'X'?

The Manual Page

Running 'man X' on Unix - and on your system if the necessary backends are installed ('man' or 'xman' is generally a good idea for terse descriptions of terms, apps and interfaces) - spits out something (cryptical) like:

NAME

X- a portable, network-transparent window system

SYNOPSIS

The X Window System is a network transparent window system which runs on a wide range of computing and graphics machines. It should be relatively straightforward to build the X Consortium software distribution on most ANSI C and POSIX compliant systems. Commercial implementations are also available for a wide range of platforms. The X Consortium requests that the following names be used when referring to this software:

X; X Window System; X Version 11; X Window System, Version 11; X11

X Window System is a trademark of X Consortium, Inc.

DESCRIPTION

X Window System servers run on computers with bitmap displays. The server distributes user input to and accepts output requests from various client programs through a variety of different interprocess communication channels. Although the most common case is for the client programs to be running on the same machine as the server, clients can be run transparently from other machines (including machines with different architectures and operating systems) as well. X supports overlapping hierarchical subwindows and text and graphics operations, on both monochrome and color displays. For a full explanation of the functions that are available, see the Xlib - C Language X Interface manual, the X Window System Protocol specification, the X Toolkit Intrinsics - C Language Interface manual, and various toolkit documents.

The number of programs that use X is quite large. Programs provided in the core X Consortium distribution include: a terminal emulator, xterm; a window manager, twm; [...] access control programs, xauth, xhost, and iceauth; user preference setting programs, xrdb, xcmsdb, xset, xsetroot, xstdcmap, and xmodmap; [...] utilities for listing information about [...] displays, xdpinfo, xlsatoms, and xprop; [...] a display server and related utilities, Xserver [...]; a utility to terminate clients, xkill; [...]

Many other utilities, window managers, games, toolkits, etc. are included as user-contributed software in the X Consortium distribution, or are available using anonymous ftp on the Internet. See your site administrator for details.

Nice tip, isn't it: On a single user system like an OS/2 workstation the 'site administrator' is: You!

3.2 The 'official' port of XFree86

The newest OS/2-port is introduced in "The XFree86/OS2 FAQ" (by Holger Veit, Sebastien Marineau):

<http://www-set.gmd.de/~veit/os2/x11os2faq.html>

3.2.1 General Information

*1.1 What is X11, X11R6.3, XFree86, XFree86/OS2?

X11, more precisely called "The \mathcal{X} Window System" is a complete window system that usually runs as the de-facto standard in Unix environments. X11R6.3 is the name of the current release (precisely, the most recent one is X11R6.4, but this is not part of the 3.X server line. XFree86 is a port of the \mathcal{X} Window System to x86-based systems. XFree86/OS2 is a particular port of XFree86 for OS/2 based systems.

1.2 Where can I find more information?

Books about X11 exist in any well-sorted CS bookstore.

Some URLs:

<http://www.x11.org/> (this has a rather large bibliography)

<http://www.xfree86.org/>

<http://set.gmd.de/~veit/os2/xf86os2.html> (XFree86/OS2)

Newsgroups:

`comp.windows.x.*` (X11 specific things)

`comp.os.os2.programmer.*` (OS/2 specific things)

Mailing list (for XFree86/OS2): see Q 1.7 [...]

*1.7 Is there a mailing list for XFree86/OS2?

Yes, read

<http://set.gmd.de/~veit/os2/xf86mail.html>

for details. [...] There is also a digest version available, read the above URL. [...]

1.16 Are there some X11 books?

Sure, ask in your book store. A user's guide is for instance:

Niall Mansfield, The \mathcal{X} Window System - A User's Guide, Addison Wesley, or The Definite Guides to the \mathcal{X} Windows System, Volume Three, \mathcal{X} Window System User's Guide, O'Reilly&Associates

The latter one is part of an eight (nine,ten?) volume documentation set for the X11 programmer. From my biased point of view of a programmer, this is the most comprehensive must-have for the serious programmer (however, it does not belong to the cheaper booksets, unfortunately)."

3.2.2 The XFree86/OS2 Infrastructure

- The hardcoded XFree86/OS2 directory structure looks like this and must not be changed:

```
$(X11ROOT)XFree86
$(X11ROOT)XFree86/bin (binaries in PATH)
$(X11ROOT)XFree86/lib (DLL's and development libraries)
$(X11ROOT)XFree86/lib/X11 (subdir's with config files)
yourhome_directory - (config files for various applications)
```

while \$(X11ROOT) normally is a drive like 'D:' and \$(HOME) is free.

- GNU File Utilities

XFree86 and most of its ports stem from Unix/POSIX; it is advisable that you download the Posix' standard utilities like rm, grep, mkdir, cp etc.

- Shells

The required shell for Rexx is CMD.EXE. While this is familiar, porting will require a shell that acts like AT&T 'sh', like the xksh. I recommend a recent version of ash/bsh, the Berkeley Unix standard shell; other ported shells often have too small command-line buffers.

3.2.3 Tip: Everblue

- Look at

"The OS/2 OpenSource Project: <http://www.netlabs.org>"

and search for the brand new project Everblue.

"Adrian Gschwend" <ktk@netlabs.org> introduces it: "Today Brian Smith released the first beta of EverBlue, the XLib-PM implementation for OS/2 which allows you to compile Xlib (Unix/Linux) applications as native PM applications.

The beta is not really usable for endusers but developers can use it to compile PM versions of X-Applications. There is just a small demo included so maybe it does not make sense to download it now if you are not a developer.

Sourcecode of this betadrop is also included in the package, just install it. If you want to support this project you can subscribe to the EverBlue mailinglist at Egroups, we still need a lot of support because the project is not yet complete. I will set up a CVS server at Netlabs as soon as we decided how to organise the structure of the source. Informations about CVS will be posted to the EverBlue mailinglist.

Everblue mailinglist: <http://www.egroups.com/group/everblue-dev>

Get the file from: (self installing exe-file)

```
ftp://ftp.netlabs.org/pub/everblue/everblue.exe"
```

3.3 And Posix, etc.?

Run e.g. on Linux

```
man 2 intro
NAME
intro - Introduction to system calls
DESCRIPTION
This chapter describes system calls.
Calling Directly In most cases, it is unnecessary to invoke a system
call directly, but there are times when the Standard C library does
not implement a nice function call for you.
Synopsis #include <unistd.h> [...]
CONFORMING TO
Certain codes are used to indicate Unix variants and standards to
which calls in the section conform. These are:
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", available at ftp://ftp.fpk.novell.com/pub/unix-standards/svid in Posts
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).
[Note: No specific emx support!]
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.
FILES
/usr/include/linux/_unistd.h"
```

Note: You are not in Linux! Nor will you use GNU glibc. IMHO GNU glibc is a sad example of investing too much work in interfaces that invite programmers to write unportable code and that unnecessarily eat up resources. But in fact they never really claimed to be Posix, only to (often arbitrarily) aim at some compatibility, while implementing things 'better'; very similar is their goal to make a 'better' platform than BSD Unix, while retaining some compatibility. EMX has:

```
emx/include/unistd.h
emx/include/io.h
```

Cf. Chapter 1.3 of the EMX LIB C REFERENCE for an overview of 'unistd.h' and 'io.h' library interfaces. Cf. also Chapter 10:

System calls are documented in /emx/doc/system.doc.

All system calls are declared in <emx/syscalls.h>. Interface routines are in the emx.a and emx.lib libraries. System call emulation routines are in the sys.lib libraries. Please do not use system calls from application programs – always use C library functions. If you need a system call, put a function into the library which issues the system call. [...] The goal is to have Unix-like system calls.

The term 'POSIX' denotes in the following all (development) platforms "*aiming at IEEE 1003.1/2 compatability*" and is not restricted to those relatively few commercial platforms that passed the official tests and paid the license fee. Please note that there is e.g. an old Linux/POSIX distribution with an 'official' POSIX label. This means: We will neglect all old broken pre-Posix Unix-like systems and encourage to completely forget about them.

Moreover if we go into details we should explain XPG, X/OPEN, Spec1170, etc. Please consult Unix tutorials here.

And to How Get Posix, etc. Defined?

Specify the desired environment (cf. emx GNU gcc manual 2.12/2.8):

```
"The preprocessor CPP defines the symbols __32BIT__ and
__EMX__. If the -Zmt, -Zmts, or -Zmtd option is given on the
GCC command line, the symbol __MT__ is defined."
"The following assertions are predefined in the emx port of
gcc:
#system(unix),
#system(emx),
#cpu(i386) and
#machine(i386)."
```

Some important gcc switches are:

'-ansi' switch to the GNU C compiler defines __STRICT_ANSI__. This is recommended, those definitions in the headers which contradict ANSI will not be included; most of the extensions are usually not necessary. Tip: Use

```
#define inline __inline__
```

to become standard conformant and to avoid error messages.

'-pedantic' switch complains loudly about non_ansi code,

'-posix' defines _POSIX_SOURCE.

If more than one of these are defined, they accumulate. For example:

```
__STRICT_ANSI__, _POSIX_SOURCE and _POSIX_C_SOURCE
```

together give you ISO C, 1003.1, and 1003.2, but nothing else:

```
__STRICT_ANSI__ ISO Standard C.
_POSIX_SOURCE IEEE Std 1003.1.
_POSIX_C_SOURCE
  If ==1, like _POSIX_SOURCE;
  if >=2 add IEEE Std 1003.2;
  if >=199309L, add IEEE Std 1003.1b-1993
  [Note: No specific emx support! Also cf.
  /XFREE86/INCLUDE/X11/Xos.h from XFree86/OS2]
```

'-v -Q' gives you important debug info, including builtin symbols and assertions, as well as a complete list of enabled optimization switches. The latter is crucial for pgcc's experimental optimizing. I've experienced internal compiler problems ('virtual memory exhausted') or unexpected program behaviour with pgcc v. 1.1.3 (gcc v. 2.92.66) and the following optimizations (if in any doubt, use the switches below). I also recommend to disable the experimental C++ feature 'rtti' and 'exceptions':

```
-fno-gcse -fno-schedule-insns2 -fno-reg-reg-copy-opt -fno-omit-frame-pointer -fno-exce
```


Chapter 4

A Silly Demo and Some Comments

Now time has come to compile some simple demo code. Here I will not add another one to the pile of introduction to ANSI C, several excellent tutorials exist. All I can express here is a kind of compliment to Eberhard Mattes for his emx development tools and libraries: There is not much to say about compiling standard C (and C++; provided you use a recent compiler version) with emx. Just start.

4.1 Read the Emx Documentation

This is just the right moment to re-read the README and INSTALL file of emx. Then compile the sample code provided by Eberhard Mattes and have a close look at the first chapters of the Emx Application Developers Guide, which introduce its philosophy. Doing both carefully might save you trouble later:

“There are three methods for creating executable files:

(E1) using `ld` and `emxbind`

(E2) using `emxomf`, `emxomfld` and `LINK386`, the program will use the `emx.dll` dynamic link library for performing system calls

(E3) using `emxomf`, `emxomfld` and `LINK386`, the program will be linked with a system call library (creating a stand-alone application or DLL)

The assembler [GNU `as`] creates a Unix-style *a.out* object file (`.o` file). When using method (E1), `.o` files created by the assembler are linked by a Unix-style linker [GNU `ld`] with Unix-style libraries (`.a` files) to create a Unix-style `a.out` file. Then, `emxbind` is used to turn this file into an `.exe` file that can be executed under both OS/2 and DOS. Using method (E1) enables *core dumps* and `fork()`. Moreover, programs created with method (E1) can be debugged using GDB,

the GNU debugger. Programs created using method (E1) use `emx` (`emx.dll` under OS/2, `emx.exe` under DOS) for system calls.

When using method (E2), the `.o` files created by the assembler are converted to *Object Module Format* files (`.obj` files). These files are linked with the OS/2 linker LINK386. The libraries are `.lib` files. `emxomfld` is a front end to LINK386 which converts the `ld` command line to a LINK386 command line. Programs created with method (E2) cannot create core dumps, cannot call `fork()` and cannot be debugged with GDB. Method (E2) works only under OS/2 and creates programs that work only under OS/2. You can use IBM's IPMD and SD386 debuggers to debug programs created with methods (E2) and (E3). Files created with method (E2) are usually smaller. The `emx.dll` dynamic link library is used for system calls. The `-Zomf` option of GCC selects method (E2).

When using method (E3), the program won't call the `emx.dll` dynamic link library. A system call library (`emx` emulator) is linked to the program. The system call library maps system calls to OS/2 API calls. Only a subset of the `emx` system calls is available with method (E3). For instance, the general terminal interface is not available. Functions which are not available or are limited with method (E3) are marked [*] in the library reference. Use a module definition file and the `STACKSIZE` statement to set the stack size. Alternatively, you can use the `-Zstack` option of GCC.'

The default stack size is 0x8000 bytes. 512 kbytes of local stack are sufficient for small tools, but too small for complex programs; pre-allocate 32 Mbytes local stack to be on the safer side. Note that all of the command line arguments and environment pointers are copied to the stack on startup!!! Small local stack-sizes are the most prominent reason for seemingly arbitrary erroneous program behaviour. (Your program code is correct: The error will disappear when you perform the usual testing in a debugger ;-).

4.2 Source with Comments

Now a trivial example to illustrate some Posix compliant `emx` interfaces. Refer to the `Emx C Library Reference` for the details about `open()`, `read()`, `write()`, etc.

(For an example how to use the more efficient ANSI streams counterparts `fopen()`, `fread()`, `fwrite()`, etc., and how to change file attributes like date and time, please cf. below: `cpfile.c` in the Chapter "Some Notes on Portability and Coding Style". It is an example for a widely portable `symlink()` replacement).

Please read the comments in the source; in fact the source has only been written to illustrate the comments and to introduce the compiler commandline frontend '`gcc.exe`'.

```

-----snip: cut here-----
/* demonstrate_EMX_and_XFree86_os2_io.c: gcc -Zbin-files

Just try to compile this primitive example with any
commercial OS/2 compiler
*/

#define TRUE 1
#define _POSIX_C_SOURCE 2

#include <unistd.h> /* Posix io and constants; contains
                    #define _POSIX_SOURCE,
                    #define _POSIX_VERSION          199009L
                    */
#include <stdio.h> /* ANSI conformant io, Posix extensions:
                    * fdopen(), fileno()
                    */
#include <stdlib.h> /* ANSI C: getenv(); Posix: environ */
#include <sys/types.h> /* Posix: for open() you need
                       sys/types.h, */
#include <sys/stat.h> /* sys/stat.h and fcntl.h */
#include <fcntl.h> /* man pages/Lib Ref. of the command say
                   which include files need to be mentioned, for
                   portability follow ANSI and Posix, Xapps should
                   conform to the similar MIT XOpen standard */
#include <string.h>

/* MIT XOpen/Posix and ANSI type globals */

ssize_t bytes_read; /* ssize_t is in unistd.h, sys/types.h */
size_t fp_bytes_read; /* size_t is in unistd.h, stdio.h */
int fd, fd2; /* File descriptors - unbuffered */
FILE *fp, *fp2; /* stdio.h - buffering */
char buffer[BUFSIZ] = ""; /* BUFSIZ is set up in stdio.h */

/* Use File descriptors - unbuffered */
static inline void
rawRead(const char *fileOrDevice)
{
#ifdef __EMX__ /* use only this for emx specific code,
              not: OS2, 32BIT, etc. */
    fd = open(fileOrDevice, O_RDONLY); /* include in low-level
                                       fd-table */
#else
    fd = open(fileOrDevice, O_RDONLY|O_TEXT); /* cf: emx C
#endif
    if (fd == -1){
        printf("Error! Reading fileOrDevice: ");

```

```

    perror(fileOrDevice);
    exit(1);
}

while (TRUE){
    bytes_read = read(fd, buffer, BUFSIZ);
    if (bytes_read > 0)
        printf("%d bytes read from %s.\n", bytes_read,
            fileOrDevice);
    else {
        if (bytes_read == 0){
            printf("End of file %s reached\n", fileOrDevice);
            close(fd); /* Remove from the file-descriptor table!
                Never mix buffered/raw i/o on the same
                file or device */

            break;
        }
        else if (bytes_read == -1){
            printf("Error! Reading fileOrDevice: ");
            perror(fileOrDevice);
            exit(2);
        }
    }
}

/* use buffering */
static inline char const *
bufferedRead(void)
{
    char const *etc_password =
        (char const*)malloc(sizeof(getenv("ETC")) +
            sizeof("TCPOS2.INI"));
    strcpy((char*)etc_password, getenv("ETC"));
    /* '/tcpos2.ini' is equivalent; cf. below
    * the section about Unix file systems.*/
    strcat((char*)etc_password, "TCPOS2.INI");

    fp = fopen(etc_password, "rt"); /* "t" is ANSI */
    if (fp == NULL){ /* read below about changing fopen(), etc. */
        printf("fopen failed to open %s.", etc_password);
        fputs("No secret file found\n", stderr);
        exit(1);
    }
}

while(TRUE){
    fp_bytes_read = fread(buffer, 1, BUFSIZ, fp);
    if (fp_bytes_read == 0)
        break;
    printf("\n%d bytes read and copied from %s.\n",
        fp_bytes_read, etc_password);
}

```

```

    }
    rewind(fp); /* go back to the start of the file */

    /* Find the descriptor associated with a stream */
    fd2 = fileno(fp);
    if (fd2 == -1)
        fputs("fileno() failed\n", stderr);

    /* Find the stream associated with a descriptor */
    fp2 = fdopen (fd2, "rt");
    if (fp2 == NULL)
        fputs("fdopen failed\n", stderr);
    fclose(fp2);
    return (etc_password);
}

    int
main(int argc, const char *argv[])
{
    const char *secret = bufferedRead();

    printf("Your secret file found in: %sn", secret);
    if (argv[1] == NULL)
        rawRead("/config.sys");
    else
        rawRead(argv[1]);

    return(0);
}
-----snip: cut here-----

```

4.3 Module Definition File - Executable

compile the example with

```
'gcc -Wall -O2 -Zomf -Zcrt.dll -Zbin-files -o demo.exe
demonstrate_EMX_and_XFree86_os2_io.c'
```

You are likely to see a LINK386 warning (we specified `-Zomf`) about a missing module definition file. This can be avoided by setting the LINK386 environment in `config.sys`:

```
set LINK386=/PM:VIO/NOIGNORECASE/NOO
```

Now, the OS2-linker as well as the `emx-GNU ld` also reads standard OS/2 module definition files, which supply the linker with useful flags. `WINDOW-COMPAT` is equivalent to the `/PM:VIO` flag, both state that your program can safely be started in an OS/2 command line window.

Note #1:

Especially important is to remember the static OS/2 stack size, so that you need to choose the right size at compile time. SYSVr4 Unix systems have a dynamic size management, so this is an OS/2 problem.

In my opinion, too small stack sizes, usually chosen with the `-Zomf` switch and its far too small default size of 32k, are the #1 reason for unexpected program behaviour or seemingly random crashes. The effect often only becomes visible if large data objects are processed. So please remember ! Don't be too shy to preallocate several Mbytes, as OS/2 will efficiently manage physical memory.

```

-----snip----- ;
;EXE.DEF: module definition file for an executable
;running in an OS/2 window or under XFree86/OS2,
;
NAME WINDOWCOMPAT NEWFILES
DESCRIPTION
  'demo programs, emx 0.9d runtime required'
; cf. emx Reference: Equal to 'gcc -Zstack 512'
STACKSIZE
  524288
; enlarge if necessary
HEAPSIZE
  524288
; enlarge if necessary
; Note: X apps coming from Unix often need huge values,
; 32 Mbytes won't hurt!
; end module definition file
-----snap-----

```

compile the example with

```
'gcc -Wall -O2 -Zomf -Zcrt.dll -Zbin-files -o demo.exe
EXE.DEF demio.c'
```

Note #2:

You must use a large initial heap size and you must not use the `-Zomf` switch if you want to make use of the `emx fork()` emulation. This Unix system call has some inherent restrictions, which often become visible with large applications, most notably X clients. Cf. the description of `fork()` and the following excerpt of the emx docs:

“The `malloc()` implementation of emx 0.9c and later can use multiple heap objects under OS/2; in consequence, the heap size is no longer limited under OS/2. However, if you use another `malloc()`

implementation which does not support non-contiguous memory allocation or if the application is run under DOS, the maximum size of the heap available for `malloc()` is fixed. [...]

The default heap size is 32 MByte under OS/2. For the `emx` implementation of `malloc()`, this is the size of the initial heap object and the minimum size of additional heap objects. If `ld` and `emxbind` are used for linking, you can change the heap size with the `-h<heap_size>` option of `emxbind`.

If `emxomf` and `LINK386` without `-Zsys` are used for linking, the heap size cannot be changed. If `-Zsys` is used, initialize the variable `_sys_heap_size` to the desired heap size:

```
unsigned _sys_heap_size = 0x4000000; /* 64 MByte */
```

This can be done in any module explicitly linked to your application. “

Chapter 5

XFree86: Hardware Gets Involved

*In the old days, when men still
were REAL MEN and wrote their
own device drivers...*

(Linus Thorvalds)

The XFree86/OS2 graphic "driver" is an application running in a separate session (process) on the local machine. The program being displayed (the \mathcal{X} client) is an application running in another separate session on the local machine or somewhere in the NET. Both communicate using the \mathcal{X} server protocol (i.e. they use the \mathcal{X} Window System Protocol specification). This one does not fit into the OSI/ISO network layer model, i.e. it is some kind of cross-over of all layers. The implementation differs considerably from system to system, but the MIT project Athena initially made it work remotely on top of TCP/IP.

5.1 Devices, \mathcal{X} Servers, Clients

5.1.1 '/dev/fastio\$', '/dev/console\$', '/dev/ptyP0\$', etc.

Many uncommon devices are being used for X.

A GUI through slip and a 9600 baud modem ain't no fun. And even an over-crowded Ethernet makes you spend your youth in front of a monitor waiting for response... So you will prefer fast local communication. For this you should compile your apps on OS/2!

This is what you will find in the current XFree86/OS2 implementation's device front:

"If you must do port I/O, use the functions from xf86sup.sys for this; this driver is there anyway, so why not take advantage of it? Common precautions for directly accessing hardware apply. Code that relies on Unix security, job control, tty/pty handling, needs major rewriting. In most cases, it is easier to make a separate file for OS/2 rather than clutter the original one with `#ifdef` clauses."

Q: What is the XF86SUP.SYS device driver good for?

"This device driver provides certain essential base functionality which is missing from standard OS/2." [...]

"When I use xconsole, there is always a warning message in the first line. What is that? You mean the text Warning: process PID=????? is accessing /dev/fastio\$, right? This is an intentional notification that some process uses the fastio device to perform access to I/O ports. It is also intentional that this message cannot be disabled. As long as the PID is the process ID of the \mathcal{X} server, everything is okay. If the PID belongs to another, unknown process, you should seriously ask the author of the program what he/she is doing with our hardware. [...]"

(The XFree86/2 faq)

So Inter-process communication uses BSD `inet SOCKETS` for remote, a `NAMED PIPE` for local communication, `pseudo-terminals` which are unfortunately not fully BSD compatible (for porting you need to look into the 'Xterm' port source code) and some other uncommon extensions, provided by 'XF86SUP.SYS'

Cf. <http://set.gmd.de/~veit/os2/xf86sup.html>

and the excerpts of the faq, reordered and abridged by me for a faster overview:

5.1.2 Concepts Related to Hardware (vio) Access and Kernel Level API 's

OS/2 API : "You may use all Dos* API functions, as long as they don't conflict with emx somehow (e.g. `_beginthread` vs. `DosCreateThread`). Stay away from `Kbd*`, `Mou*`, `Vio*` calls - none of those

will do what you expect, but you can severely influence the operation of the X11 system. [...]"

Hint: Use instead ANSI C ('stdio.h') (or POSIX ('unistd.h')) interfaces!

"3.11 Can I use another shell like 'bash' in place of CMD.EXE in an xterm? Most shells are subtly incompatible with the PTY mechanism used in xterm. A symptom may be that they won't work either if you redirect console I/O to a COM terminal which should be possible. bash is such an incompatible shell. See also Q 3.12 and Q 3.13.

Meanwhile, there are special ports of tcsh and ksh which work in an xterm.

3.12 I use 4OS/2 (or some other shell) as my shell. Will it work in xterms? 4OS2 has a number of misfeatures, politely spoken, in an xterm. We attempt to fix this in the future. For now, don't use it in an an xterm. If 4OS2 or another shell is in your OS2_SHELL or SHELL variable, please add an environment variable named X11SHELL to point to a valid CMD.EXE path, e.g. SET X11SHELL=D:OS2CMD.EXE in your CONFIG.SYS. This will override the OS2_SHELL or SHELL setting which is used otherwise. 3.13 Why does OS2MORE.COM give a SYS0447 in an xterm? This is a program which silently assumes it has access to the keyboard and the screen: it uses functions from KBDSCALLS, MOUCALLS, or VIOCALLS. There are more programs of this kind, for instance: most *.COM in os2, ATTRIB.EXE, BACKUP.EXE, CACHE.EXE, EAUTIL.EXE, FIND.EXE, HELPMMSG.EXE, LINK.EXE, LINK386.EXE(!), PATCH.EXE, PSTAT.EXE, REPLACE.EXE, RESTORE.EXE, SETBOOT.EXE, SORT.EXE, SPOOL.EXE, SYSLEVEL.EXE, TEDIT.EXE, TRACE.EXE, UNPACK.EXE, XCOPY.EXE, XDFCOPY.EXE, but also unfortunately TELNET and FTP, and some others; infact, almost any 16 bit application is a possible candidate. Note that CMD.EXE is an exception of the rule: it is largely still 16 bit, but is clean for XFree86/OS2 use.

You can use the EWS utility EXEMAP.EXE to check if these DLLs are linked in.

Note: the existance of these DLLs does not mean that the program does not work at all, actually EMX.DLL could call KBD API functions but usually does not in the X11 environment. However, certain unexpected effects may occur in an xterm. I'll try to find a workaround for this in the future, but don't count on this. 3.14 FTP seems to work in an xterm, but it does not hide the password? This is a side effect of what was described in Q 3.13. So actually FTP does NOT work. [...]

5.2 Inter Process (Session) Communication

Now some concepts and short hints related to the NAMED PIPE for local communication with the \mathcal{X} server. Emx unfortunately does not yet have a user friendly Posix compatible `mkfifo()` interface, this is due to the still missing Posix compatible file system. You can use the native `dos*npipe()` interfaces of OS/2 to create `/PIPE/*` devices, then register file handles via `_imphandle()` to use them with emx library functions as well as with the native OS/2 kernel api. Note: Their behaviour is not Posix compatible; cf. for details OS/2 api reference manuals.

2.52 What does the error message "`_X11TransOs2OpenClient: Open server pipe /PIPE/Xxf86.0 failed`" mean?

This is for the local named pipe communication the same problem as Q 2.31 is for the network connection. It basically means: the server crashed for some reason, and now a client, e.g. xterm cannot connect to the server. The reason is probably hidden somewhere else in the XF86Config file, or some other setup problem (e.g. network configuration). [...]

2.31 What does "`SocketINETConnect() can't connect: errno = 65`" mean? What does "`SocketINETConnect() can't connect: errno = 61`" mean? These messages **can** point out a network installation problem, in many cases they are secondary errors, though, and are really caused by a different problem. Nevertheless, you should ensure that your network is setup correctly. [...]

3.8 How can I enable authorization for host `foo.edu` without using `xhost`?

Create the file `X0.hosts` in `XFree86libX11`. To enable connections from a certain host, add in a line containing `inet:hostname` to the file. For example, your file may look like this: `inet:foo.edu, inet:friendly.host.edu ..` Note that the `X0` refers to the zero in the display name `os2systemname:0`, i.e. in the case of Q 3.6 this would become a `X1.hosts` then. [...]

5.3 Emx and device driver programming (hardware devices)

Q: Can emx be used to generate 16 bit code for programming OS/2 device drivers?

No!. :

(Holger Veit): "as gas lacks the ability to produce arbitrary segment types beyond 32 bit `.text/.data/.bss` segments. It can generate 16 bit instructions in a 32 bit segment (prefixed with `0x66/0x67`), though."

On Wed, 13 Oct 1999 17:58:35 +0100, Csaba Raduly wrote:

5.3. *EMX AND DEVICE DRIVER PROGRAMMING (HARDWARE DEVICES)* 63

"But you can write device drivers in 32-bit using some (already existing) thunking mechanism

Check out

`/pub/os2/system/drivers/filesys/32drv170.zip`

on hobbes."

IBM distributes the device driver devel package (including compiler) for free!
You may want to look at

`http://rover.wiesbaden.netsurf.de/~meile/warpstock/wsp02/en/index.html`for the details...

Chapter 6

Building Libraries

6.1 Explaining Some Library Concepts

6.1.1 Dynamic Link Libraries (DLL's) Versus Static Libraries

Dynamic link libraries are great, but *ain't easy*.

6.1.1.1 Static Libraries (*.a or *.lib)

Historic pre-Posix systems of the Unix family, using the *a.out* binary format, allowed static linkage only:

Common functions could only be archived for faster linking with the librarian `ar` and the librarian indexer `ranlib` (or `ar -s`), thus creating *a.out* archives. Most prominent is the standard C library system call library (`libc.a`), which every program needs. It is linked in automatically. This is equivalent to `c.a` (*a.out*) and `c.lib` (*omf*) of `emx`, plus some additional system specific helper libraries. Conversion is possible, cf. `emxomf` docs.

A difference is the math library (`libm.a`): Traditional Unix uses a separate `libm.a`. It must be specified explicitly as last entry of the linker command line (`-lm`).

`Emx` automatically statically links those functions in, - in fact they are a part of `libc.a` -, whereas `libm.a` is only a dummy. If you use `-ZcrtDLL`, the equivalent import library versions for implicit dynamic linkage are used instead; more details later.

6.1.1.2 Shared Objects (*.so)

AT&T Unix System V Release 4 (*SYSVr4*) introduced the new *ELF* binary format, which allowed for the first time an efficient and flexible use of shared global functions and variables (shared objects; `*.so`).

This is achieved by creating shared read-only *code* and special shared read-write *data* segments, which can be used by multiple processes and are now better supported by the new binary format. The global emx picture is here with a grain of salt nearly the same as the one for Unix; *omf* binary format is potentially somewhat more complex, though. A process (session) consists mainly of the following memory sections:

- Fixed **.text**: machine language commands and constants,
- variable global **.data**: global variables of your program and libraries taken from the heap
- and last but not least **.bss**: a stack pile, where your processor temporarily jots down it's masses of 'telephone notices' so that it can get them back in the right order; the poor piece of silicium cannot keep everything constantly in mind, aka registers.

The stack usage depends on the calling conventions (parameter handling) of your programming language. You can discriminate the 'system' calling convention part for system calls and the 'user' mode for application functions.

All are assembled to relocatable object code - which still contain unresolved symbols - or data segments and later, after symbol fixup, moved to their final memory location by the OS's linker and the executable loader, respectively. *A.out* segments need in addition a small *a.out-loader*, which is added by *emxbind* or they need to be converted by *-Zomf* and *emxomf* to native *object module format* before linking with *link386*.

6.1.2 Shared and Dynamic Libraries (TOG Spec1170)

Don't confuse the following, just because on OS/2 and on *ELF* systems they use the same file type, i.e.: **.so* or **.DLL*, respectively:

Shared libraries are something implicitly linked against the main executable (i.e., more precisely forwarded to it by import libraries, for systems that use this fixup technique, respectively) and loaded at the start-up of the executable.

Dynamic libraries are *not* linked to the main module, but loaded at the run-time, if need be. Any systems conforming to "*The Single UNIX Specification version 1/2*" from *The Open Group*, often also referred to as the former *Spec1170* by *X/Open Ltd.*, can use the:

`dlopen()` call to open dynamic libraries at the run-time in a unified manner and can call

`dlsym()` to actually resolve symbols, if needed.

Ports implemented by Alexander Mai, Andreas Kaiser, or an almost identical implementation copyrighted by someone else exist.

6.2 Exporting symbols from a DLL (OS/2 OMF)

Unlike most Posix systems, OS/2-emx does not export automatically any function or global variable. You must write a "module definition file", listing exported symbols. This gives you the chance to make sure that all entry points are identified by ordinal numbers, which have to be *identical* for new library revisions (binary compatibility). You can add new exports at the end of the list, though. If you don't make sure that your entry-points are identical among library revisions and that the called functions and exported global variables or constants do exactly what they should do, you will lose binary compatibility and you will have to relink all applications using the dll. After linking the dll this "module definition file" is then passed to the 'emximp' tool in order to create an "import library" (link library) that contains necessary fixups for the linker to resolve the exporty dll symbols when linking you application against this dll.

"EXPORTS Make functions and variables visible outside the .exe or .dll file. All entry points of a dynamic link library must be exported using EXPORTS. Exporting entry points of .exe files is less common. " (Emx docs).

The 'emxexp' tool as well as studying the library's public header file may help you to identify which symbols are worth to be exported for later linker fixup, when compiling your client application.

Hints:

Use short names (symbols) in a library, especially with C++. The omf linker (link386.exe) has a symbol length restrictions (512 characters). Else you'll probably see multiply defined symbols linker errors. Some more special tips at the end of this chapter.

6.3 'dynamic library' and 'shared object' (*.so) versus DLL

You'll encounter a 'dynamic library' that differs from a 'shared object' (*.so), if you try to port recent apps like 'dia', that come from Posix/SYSVr4-Unix:

"Dia plugins are all dynamic libraries, so the program stops working once you start using NONAME keyword."

Shared libraries - they are commonly called shared objects for clones of SYSVr4-Unix (which introduced the ELF-object module format) - are something implicitly linked against the main executable and loaded at the start-up of the executable. It has been described above how this is achieved explicitly by linking against import libraries created from module definition files.

Dynamic libraries are NOT linked to the main module but loaded at the runtime. Don't confuse them, just because on OS/2 and on ELF systems they are the same file. Any systems conforming to "The Single UNIX Specification

version 1/2" from 'The Open Group' can use the `dlopen()` system call to load dynamic libraries at the run-time in a unified manner and call `dlsym()` to resolve symbols. Ports of this special mechanism are available separately; `emx` does not provide the `dlopen()` and `dlsym()` system calls.

6.4 Demo

Here a demo (the common `libxpm` library); cf. the port's diff by H. Veit. This `xpm` library revision is e.g. linked into `xforms.dll` rev. 0.88.

Warning: Different `xpm` versions are incompatible. Apparently random crashes are often a sign that you mixed `xpm` versions and/or headers. You cannot mix versions! Use only the version your toolkit libraries have been tested and/or linked with. Headers are incompatible among versions, too.

Note: In `XFree86-4.0`, `libXpm` will be now finally integrated and frozen.

```

;Xpmos2.def: sample module definition file for an X dll
;under XFree86/OS2, linked with '-Zcrt.dll'
; initialize shared objects per instance
LIBRARY XPM INITINSTANCE TERMINSTANCE
DESCRIPTION "X11R6 XFree86 libXPM xpm.dll for OS/2 EMX09C VERSION=7 (xpm3.4g)"
;
;reduce memory fragmentation (for larger and less frequently used library
;code save memory and use the 'CODE LOADONCALL' linker flag instead)
CODE
    PRELOAD
;
;.text objects are globally shared memory, .data is individual to any client
;process
DATA
    MULTIPLE NONSHARED
;
;is this really needed here? Well, at least doesn't hurt.
STACKSIZE 65536
;
;exported client interface symbols for linker fixup, 'NONAME' reduces table
;size, helps that you link only with binary compatible libraries, else
;you'll experience immediate errors/crash (yes, we'll go the hard way ;-)
EXPORTS
    XpmCreatePixmapFromData @ 1 NONAME
    XpmCreateDataFromPixmap @ 2 NONAME
    XpmReadFileToPixmap @ 3 NONAME
    XpmWriteFileFromPixmap @ 4 NONAME
    XpmCreateImageFromData @ 5 NONAME
    XpmCreateDataFromImage @ 6 NONAME
    XpmReadFileToImage @ 7 NONAME

```

```

XpmWriteFileFromImage @ 8 NONAME
XpmCreateImageFromBuffer @ 9 NONAME
XpmCreatePixmapFromBuffer @ 10 NONAME
XpmCreateBufferFromImage @ 11 NONAME
XpmCreateBufferFromPixmap @ 12 NONAME
XpmReadFileToBuffer @ 13 NONAME
XpmWriteFileFromBuffer @ 14 NONAME
XpmReadFileToData @ 15 NONAME
XpmWriteFileFromData @ 16 NONAME
XpmAttributesSize @ 17 NONAME
XpmFreeAttributes @ 18 NONAME
XpmFreeExtensions @ 19 NONAME
XpmFreeXpmImage @ 20 NONAME
XpmFreeXpmInfo @ 21 NONAME
XpmGetErrorString @ 22 NONAME
XpmLibraryVersion @ 23 NONAME
XpmReadFileToXpmImage @ 24 NONAME
XpmWriteFileFromXpmImage @ 25 NONAME
XpmCreatePixmapFromXpmImage @ 26 NONAME
XpmCreateImageFromXpmImage @ 27 NONAME
XpmCreateXpmImageFromImage @ 28 NONAME
XpmCreateXpmImageFromPixmap @ 29 NONAME
XpmCreateDataFromXpmImage @ 30 NONAME
XpmCreateXpmImageFromData @ 31 NONAME
XpmCreateXpmImageFromBuffer @ 32 NONAME
XpmCreateBufferFromXpmImage @ 33 NONAME

```

Here are linker flags I recommend for larger, less frequently used dll's. Your mileage may vary. Test on systems with reduced memory and slow hd, e.g. notebooks (the excerpt comes from a def file that I used to link the X11.dll):

```

; X11.def (created by emximp)
LIBRARY X11 INITINSTANCE TERMINSTANCE

DESCRIPTION
    'X11.DLL (MIT XStandards Library for emx-port of XFree86).'
```

CODE

```

    LOADONCALL
```

DATA

```

    LOADONCALL MULTIPLE NONSHARED
```

STACKSIZE ;is this really useful?

```

    0x80000
```

HEAPSIZE

```

    0x160000
```

```

EXPORTS
    XSaveContext                @1  NONAME
    .... etc.

```

6.5 How to Link Your DLL

I recommend the following simple (unorthodox, but nearly 'foolproof') procedure:

Use the static archive provided on the net or use the Makefile (normally created by Imake, but today more and more by the OS/2 port of autoconf; details later) to make the static archive `foo_s.a`. Then:

1. `'emxomf -s -l foo_s.a'` (convert to native omf format)
2. `'emximp -o foo.a foo.def'` (create import library for later ld linker fixup)
3. `'emxomf -s -l foo.a'` (convert to native omf format for link386)
4. `'gcc foo.def -o foo.dll -Zomf -Zdll -Zmtd -Zsysv-signals -Zbin-files -Zlinker /EXEPACK:2 -Zlinker /NOI -Zlinker /NOL -Zlinker /NOD -s -fstack-check -lfoo_s -lmore_foo_libs_here -lXmu -lXt -lSM -lICE -lX11 -lXext -lbsd -lsocket'`

This produces a packed DLL. While some linker flags aren't really necessary, they won't hurt, unless you use OS/2 2.x (use '/EXEPACK' if in doubt). Isn't the commandline nice?

Linking Order

Linking order is crucial! The traditional C-style linker must find the fixup for still unresolved symbols in the same library or in another one which is `_following_` on the command line. You might even need to link in rare cases like: `-lfoo1 -lfoo2 -lfoo1`

where `foo1` needs symbols exported by `foo2` and `foo2` needs symbols exported by `foo1`.

6.6 Library Debugging

For a debuggable library compiled with `'-g'` you'd use the same line (except: `'-Zomf -s'` and the `'-Zlinker blurb'`, which are specific to `link386`).

6.7 Building a C++ DLL

Cf. emx docs:

"You should not use the GNU C++ libraries (`libg++` and `libstdc++`) if you create a DLL (unless you are building a stand-alone DLL or a private C runtime DLL) as there is not yet a DLL version of these libraries.

As creating the module definition file for the DLL by hand is quite boring, you should let your computer do it: see the description of the `emxexp` tool for details. The `sign` sample of `emxample.zip` uses `emxexp` to create the module definition file.

To call the constructors and destructors for static objects in the DLL, make sure that the `_DLL_InitTerm()` function of the DLL calls `__ctorctorInit()` and `__ctorctorTerm()`. You should call `_CRT_init()` before `__ctorctorInit()` to ensure that the C library has been initialized for use by the constructors, even if OS/2 happens to initialize the DLLs in a wrong sequence. " [The default function, being linked automatically with `-Zd11`, should suffice.]

6.8 Some Special Tricks for libraries

6.8.1 Instead of using preprocessor macros, you can use import libraries

Beware: Redefines of functions are debugging headaches: The actual code won't be explicitly in the sources; you will forget, that you redef'd!

(On Mon, 26 Apr 1999 08:59:44 +0900, Shigeru Miyata wrote:)

Instead of using preprocessor macros, you can use import libraries for the purpose in some cases. If the prototypes of the functions are declared somewhere, you can run `emximp` to create `.a/.lib` on the module definition file like this:

```
LIBRARY emxlibcm
EXPORTS
strcasecmp @219
strncasecmp @225
;getcwd @779
;chdir @764
;lstat @561
```

This is the generic syntax:

```
EXPORTS entryname [=internalname] [@ordinal [RESIDENTNAME|NONAME]]
[...]
```

entryname is the name of the function as made visible outside of the .exe or .dll file. **entryname** is always converted to upper case. **internalname** is the name of the function as defined in your program. If **=internalname** is omitted, it is assumed to be identical to **entryname**. **internalname** is case sensitive. Exported functions not only have a name (**entryname**), they also have an *ordinal number*, the position within the name table. Using ordinal numbers when importing saves space and is supposed to be faster.

Example:

```
EXPORTS
my_qsort=qsort1 @1 RESIDENTNAME
```

Use **NONAME** to avoid putting **entryname** into the name tables.

Notice also that you can concatenate import libraries by `ar/emxomfar`. So if you don't like OS/2 specific libraries to be passed to the linker like `-lmmmap -lshm ...` Then you can make `iberty.a/iberty.lib` for OS/2.

6.8.2 Debugging huge DLL's

(On Fri, 15 Oct 99 08:59:15, Asbjoern Pettersen wrote:)

>I'm no popuplog expert. Where do I find more info about it ?

IBM service pac CDs contain a Redbook ("The OS/2 Debugging Handbook, #0.6"), if you really want to undergo the pain of low level (Intel architecture) debugging.

Can't you just compile a.out (no `-Zomf` and `-g`, this links also the dll's with `ld.exe`, but you might have to link the *.o files directly, linking against archives *.a sometimes miss symbols despite `ar -s`) and use `pmgdb` or `gdb`? Moreover you have then the luxury of nice core dumps, that poor non-emx developers can only envy!

So you ought to provide debuggable a.out binaries. What you can do (and what I've done to build L^AT_EX dll's) is to compile without `-Zomf` and with `-s`. After that you'll compile an unbound a.out symbol file (same procedure as before, just leave out the '.dll' suffix from `-o foo.dll`. If need be, you can always feed the debugger this symbol file, while the dll has a reasonable size.

>> Well surely I can create "-g" dll monsters ;) But there is a problem, `ld.exe`

>> refuses to build dll from my .def files

>> complaining about some symbols, etc.. while `link386` is up and ok.

>>`link386` has `/NOI` flag, while `ld` does not have an equivalent.

>I haven't tested but modifying the line 47 of `emx/src/emxbind/export.c`

>may help.

>Anyway if I were you, I would compile only the relevant object into

```

>the exe with debugging on. It seems you haven't created the map file
>for glib12.dll Try using exemap as follows. Since I don't have
>gtk-1.2.5 here, the addresses below are not exact.
>Suppose you have a SIGSEGV in GLIB12.DLL 0001:00000e08
>exemap glib12.dll|grep "1 00000"|sort +2
>Now you will have something like
> 6 1 00000038 03 Entry32 EXPORT GLOBAL (G_ARRAY_NEW)
> 4 1 000000F8 03 Entry32 EXPORT GLOBAL (G_ARRAY_FREE)
> 3 1 00000168 03 Entry32 EXPORT GLOBAL (G_ARRAY_APPEND_VALS)
> 7 1 000001A8 03 Entry32 EXPORT GLOBAL (G_ARRAY_PREPEND_VALS)
> 5 1 00000208 03 Entry32 EXPORT GLOBAL (G_ARRAY_INSERT_VALS)
>[...]
> 27 1 00000C74 03 Entry32 EXPORT GLOBAL (G_CACHE_NEW)
> 24 1 00000DD4 03 Entry32 EXPORT GLOBAL (G_CACHE_DESTROY)
> 25 1 00000E34 03 Entry32 EXPORT GLOBAL (G_CACHE_INSERT)
> 28 1 00000EF4 03 Entry32 EXPORT GLOBAL (G_CACHE_REMOVE)
> 26 1 00000FB4 03 Entry32 EXPORT GLOBAL (G_CACHE_KEY_FOREACH)
>(The table above is that of glib12.dll v1.2.2 for emx/OS2)
>Now you know that the crash occurs in somewhere between "1 00000DD4"
>and "1 00000E34". Hence it is in G_CACHE_DESTROY @24 which is in
>gcache.obj
>It is a good bet that the problem is in fact enclosed in gcache.c
>and a few more sources. Then you don't have to build whole the
>library with debugging symbols.
>Add gcache.c to the source of scriptfu.exe, build and run it under
>gdb.

```

6.8.3 Length limitation for symbol names in OMF-Format

(On Sun, 17 Oct 1999 14:34:21 +0200 (MET DST), Stefan Neis wrote):

```

> Hi,
>> 1) When i compiling and linking programme with key -Zomf, i get in result
link
>> error message: symbol defined more then once.
>> 2) When compiling and linking without omf libs (with ld) all ok.

```

>>

>AFAIR, there's some length limitation for symbol names in OMF-Format.

>gcc tends to generate really huge names. Especially for template

>instantiations a very large part at the beginning of two different

>instantiations might be identical, so the automatic truncating will give

>you the same name for different symbols.

>'nm' might be helpful to verify, if this is your problem.

>If it is, you can't do very much about it, besides praying for

>either IBM removing this stupid length limit or

>somebody modifying gcc to generate shorter unique symbols :-)

Note: `pgcc` already contains an experimental symbol name compression extension.

6.8.4 Specify the name without the extension

(On Wed, 20 Oct 1999 17:28:36 +0200, Alexander Mai wrote):

>Potential problem might be the DLL name: IIRC one can either use

> foo

>or

> z:barfoo.dll

>to access foo.dll (if in LIBPATH).

'foo.dll' generally does not work. It works if and only if it is already loaded into memory by other processes (or the process itself). If you want to use DLL as a dynamic library, specify the name either without the extension or by the fullpath with the extension.

6.8.5 C++-DLL's

(Compiling qt apps under EMX; S.Miyata)

" -Qt 1.x apps must be compiled either with

emx09c without special flags

or

emx09d with the flags `-fno-rtti -fno-exceptions`

-Qt 2.x apps must be compiled with emx09d without special flags.

-Zomf is not recommended for Qt 2.x, the link may fail.

Now the problem with stdcpp.a turns out to be a bug in emx09d fixpack2 (missing exports in emxlibc?.dll). Using stdcpp.a from emx09c might work, but it is the last resort. I cannot guarantee the result.

Also stdcpp.a is compiled with the default flags in gcc 2.8 with '-frtti -fexceptions'. Only fix is to recompile stdcpp.a yourself with '-fno-rtti -fno-exceptions' if you want to use it with Qt 1.x"

On Mon, 1 Nov 1999 16:25:35 +0100, Alexander Mai wrote:

>As long as things are rather simple, small and free of 'exceptions'
 >everything seems to be fine. e.g. fltk rarely causes problems.
 >And problems aren't solved in current egcs/gcc. It fails for valid
 >code and in addition it fails for broken code which used to compile
 >for years. It doesn't matter whom to blame for this,
 >but from a user's point of view, it's a bad situation.
 >Also it makes clear that an EMX 09e/ egcs 2.95.x wouldn't
 >solve all problems (if any).

I recommend not to use exceptions, as they are slow and often buggy with all current compilers. So the idea to recompile 'stdcpp.a' might often be the only safe solution. Emx docs recommend not to link 'stdcpp.a' into a dll, as binary compatibility is hard to achieve then.

6.8.6 Multiply defined symbols error with C++-DLL's

I observed a wierd link386.exe bug wit C++-DLL's:

Linking against a static module/library (*.o/*.a) only works, if the system default libraries, which are passed automatically to the linker by the emx frontend, do not contain fixups for already resolved references. I.e.: Your mylib.lib contains your own version of foo::foo() and hence the symbol 'wierd-C++-symbol-name-for-foo::foo-function'.

But `foo::foo()` is already defined in some standard library, which is automatically passed to the linker. Normally this is no problem, because the linker will only use the first reference and ignore the second one, so that you can safely override the standard version with your private function.

But for some reason it behaves different for certain C++-symbols and tries to link against both contradicting function definitions. So you must temporarily disable the passing of the second symbol. To achieve this you may create a modified import library, which does no more contain the symbols you want to override.

6.8.7 Sys 0008

On Fri, 12 Nov 1999 16:44:44 +0100, Holger Veit wrote:

>> Hi!

>>

>> Today when I run xchat in my \mathcal{X} with panel and GNOME icq applet I received approx.

>> such message:

>>

>> Sys 0008: ZVT Not enough memory to run program. I checked theseus I had 30Mb of

>> RAM free.

>

>I have seen this before with memory intensive programs, or debuggable X11 DLLs

>(very large). Another resource is probably exhausted, such as the space

>available for DLLs (64MB) or the process heap (about 300MB).

So, respect DLL resource limits. Process heap limits may be lifted in future versions

of OS/2, meanwhile one must live with the restriction of of 512 MB per process address

space.'

Chapter 7

Imakefile or configure, that's the question!

Traditionally \mathcal{X} apps come with an Imakefile to use the old tool available for virtually all official X11R6 platforms (and some others). According to a set of rules and system-specific templates, imake generates a giant Makefile for Unix-style make. OS/2-emx and XFree86/OS2 then uses the GNU make utility for conditional compilation and linkage. Unfortunately this one differs from original AT&T-style make especially in the handling of libraries (archives). If you experience problems, please cf. the original documentation, available in GNU INFO-format.

The FSF GNU project uses a different suite of test scripts called `auto*` tools, written for Unix: `autoconf`, `autoheader`, `automake`, ... An experimental and useful, yet - due to the different file system, kernel and API design necessarily imperfect - port to OS/2 exists.

7.1 A short overview: Autoconf vs. Imakefile (by Hung-Chi Chu <hcchu@r350.ee.ntu.edu.tw>)

I would like to share my experience about `autoconf` and `Imakefile`:

- `autoconf/emx` is not perfect, especially on Makefiles generated automatically. I don't like to modify any auto-generated files since they are "monsters" and the modification will be overwritten when the next `configure` runs.

Tip #1 (<miyata@kum.kyoto-u.ac.jp>): "It also helps to edit `config.status` and run it, instead of running `configure` again."

- `autoconf/emx` is nearly perfect for generating `config.h` except that occasionally I need to add/modify a few lines in `configure.in` for searching some special functions in add-on libraries [Cf. the list of ported libraries in the faq].

Tip #2: "Read below about some hackish `#define`'s that simulate Unix API interfaces. Add necessary `#include` and `-lFOO` statements to the test scripts."

- for other auto-generated and non-intermediate files which have path definition in them, such as the script `'gtk-config'`, write a new file called `*.os2`; substitute the drive letter with `#{X11ROOT}` et al.

Tip #3 (<http://set.gmd.de/~veit/os2/xf86os2.html>, by Holger Veit <Holger.Veit@gmd.de>):

[...] XFree86/OS2 stores all configuration files in a common directory. This is:

`X11ROOT:\XFree86\lib\X11`, where `X11ROOT` is an environment variable which contains the drive letter the directory tree is located.

The common places like `/usr/X11`, `/usr/X11R6`, `/usr/XFree86`, `/etc` which are used for XFree86 in various Unix flavors, are not supported [...]. It is also there where you should put your `XF86Config` file, the color database (`rgb.txt`), the host authorization files (`X0.hosts`), and various other files. Unix does not have drive letters, so a file (resources, fonts, config, etc.) path starts with `'/'` normally. [...]

To prepend the `X11ROOT` part to a file path, a special function named `__XOS2RedirRoot` exists which you are supposed to use in these cases, rather than an insane own solution. The prototype of this function is:

```
const char *__XOS2RedirRoot(const char *pathname);
```

- `Imakefile` is neat for X/2 libraries and programs. It is analogous to `Makefile.am` for `autoconf`. It is not difficult to write an `Imakefile` from `Makefile.am`. For non-X lib/prog, it is also simple to rewrite a neat `Makefile` for `os2` (ex. `Makefile.emx`) from `Makefile.am`.

- The effort of rewriting `Imakefile/Makefile.emx` is almost at finding all `DEFINES` for `gcc`. This can be easily done by searching `-Dfoo` in `Makefiles` generated by `configure`.

- Following are the main steps I apply to do port:

1. use `autoconf/emx` to generate `config.h`, `Makefile` et al.
2. for the first version:
 - (a) rewrite `Imakefile/Makefile.emx` from `Makefile.am`
 - (b) check some auto-generated and non-intermediate files. Modify and rename them as `*.os2` if necessary
 - (c) for dynamic library, make a module definition file `*.def` and
 - (d) a response file `*os2.rsp` (necessary for `Imakefile`)
3. for the following versions:
 - (a) apply the patch of the previous version

- (b) `diff -u previous_version/Makefile.am this_version/Makefile.am`, and if necessary modify `Imakefile/Makefile.am`
 - (c) `diff -u` of any other special `*.in` files and modify the corresponding `*.os2`
 - (d) for dynamic libraries, check `*.def` and modify it if necessary
4. for `Imakefile`: `xmkmf`; `make Makefiles`; `make` or `xmkmf -a`; `make` for `Makefile.emx`: `gmake -f Makefile.emx all`
 5. `make install` and make a patch file by `diff -ruN` if all work well
 6. make the zip package. :-)

7.2 Preliminary steps

The following is meant to overcome some simple, nevertheless annoying problems and to make things work faster: Quick and simple fixes, some dirty stolen tricks.

7.2.1 RTF docs

1. Read the `README` file! Or should it better be called `DON'TREADME` (psychology?)
2. Read the `INSTALL` file!
3. Read any additional info files from the program authors. Many problems occurring under special UN*X flavours have a similar counterpart on OS/2 (others haven't, but this shall be a simple guide :-)
4. Read the `/XFree86/lib/X11/config/README` and `/XFree86/lib/X11/config/host.def` files, for a short introduction to adapting `Imake` templates. Excerpts introducing some important concepts:

“ The easiest way to write an `Imakefile` is to find another one that does something similar and copy/modify it! To change any of these variables, edit the `site.def` file.

`Imake.tmpl` provides defaults for the following variables: [...; cf. your current version of `README`]. `Library.tmpl` provides defaults for the following variables: [...; cf. `README`]. `Server.tmpl` provides defaults for the following variables: [...; cf. `README`]. `Threads.tmpl` provides defaults for the following variables: [...; cf. `README`]. An `<os>Lib.rules` file provides defaults for the following variables: [...; cf. `README`]. An `<os>Lib.tmpl` file provides defaults for the following variables: [...; cf. `README`]. The following variables are used by some part of the tree: [...; cf. `README`].

Make Variables The following make variables are used by imake rules and may be set in an individual `Imakefile`: [...; cf. `README`]. Many other make variables are set up by the imake config files and can be used in an `Imakefile`. The easiest way to discover them is to look at the `Makefile` generated by an empty `Imakefile`.

Comments Use C comment syntax in an `Imakefile` for comments that should not appear in the resulting `Makefile`. Use `"XCOMM"` at the start of each line to produce a comment that will appear in the `Makefile`. (The `"XCOMM"` will be translated into the `Makefile` comment character `"#"` by imake.) Do NOT use `"#"` as a comment character in `Imakefiles`; it confuses the C preprocessor used by imake on some systems.

Imake variables Don't abuse the variables in `X11.tmpl` that describe particular pieces of X by using them to describe your own subsystems. Instead, create new variables that are defaulted using `Imake.tmpl` variables. [...]

Common Rules Here are some of the common rules for building programs. How to use them is described in `Imake.rules` and in the O'Reilly book "Software Portability with imake."

Basic program-building rules

All of these except `NormalProgramTarget` also generate rules to install the program and its manual page, and to generate dependencies.

SimpleProgramTarget Use if there is only one program to be made and it has only one source file.

ComplexProgramTarget Use if there is only one program to be made and it has multiple source files. Set `SRCS` to the names of the source files, set `OBJS` to the names of the object files, and set `DEPLIBS` to the libraries that this program depends on.

ComplexProgramTarget_1 Like `ComplexProgramTarget`, but uses `SRCS1`, `OBJS1`, and `DEPLIBS1` and can be used with `ComplexProgramTarget_2` and `ComplexProgramTarget_3` to build up to three programs in the same directory. Set `PROGRAMS` to the programs built by all of these rules. For more than 3 programs, use `NormalProgramTarget` for each.

ComplexProgramTarget_2 Use after `ComplexProgramTarget_1` for the second program in a directory. Uses `SRCS2`, `OBJS2`, and `DEPLIBS2`.

ComplexProgramTarget_3 Use after `ComplexProgramTarget_2` for the third program in a directory. Uses `SRCS3`, `OBJS3`, and `DEPLIBS3`.

NormalProgramTarget Build a program. Can be used multiple times with different arguments in the same `Imakefile`.

```
[...]
Imakefile for directory with subdirectories

XCOMM sample Imakefile for subdirectories
#define IHaveSubdirs
#define PassCDebugFlags CDEBUGFLAGS="$(CDEBUGFLAGS)"
SUBDIRS = list of subdirs ...
MakeSubdirs($(SUBDIRS))
DependSubdirs($(SUBDIRS))

[...]"
```

The following may be useful, too.

7.2.2 Some additions to /XFree86/lib/X11/config/os2.cf and os2.rules

SMiyata <miyata@kusm.kyoto-u.ac.jp> pointed out in posting to the XFree86/OS2 mailing list:

As "far as possible, the changes must be confined to Imakefiles and definition files bundled in the application. The settings in `os2.cf` and `os2.rules` are shared by other applications, and hence may cause conflicts in them. Also, the changes to the template files are local to your setting, thus, although it may help to build binary on your system, it is not suitable for porting, unless they are reflected in the distribution of XFree86 itself.

There are indeed the cases that modifications must be made on template files coming with XFree86 rather than Imakefile and definition files coming with the applications [those often have uncommon names like `local.cf` or `machinefile`, etc.]:

Akira Hatakeyama <akira@sra.co.jp> reported elsewhere that, in order to build `Xvnc` on OS/2, the lines

```
#define BuildXKB YES
#define BuildLBX YES
```

in `os2.cf` must be modified to

```
#ifndef BuildXKB
#define BuildXKB YES
#endif
#ifndef BuildLBX
#define BuildLBX YES
#endif
```

since, else, the settings in `vnc.def` is overwritten by those of `os2.cf`"

My proposed changed `/XFree86/lib/X11/config/os2.cf`:

```

#define OSName          OS/2
#define OSVendor        IBM
#define OSMajorVersion  3
#define OSMinorVersion  0

/*
 * C library features
 */
#ifdef __EMX__

/* set this to what you like (almost :-)
 * OS/2 does a nonstandard bootstrap of imake
 */
#define BootstrapCFlags  -DBSD43

/* X_WCHAR and X_LOCALE are set in Xosdefs.h */
#define StandardDefines  /**/
#define XawI18nDefines  -DUSE_XWCHAR_STRING -DUSE_XMBTOWC

/* This will redirect everything to /XFree86
 * Care will be taken that programs that need a file
 * from this tree will add the environment variable
 * X11ROOT to the search path so you can redirect the
 * stuff to a different drive
 */
#ifdef ProjectRoot
#undef ProjectRoot
#endif
/* This will be mangled with a drive letter in most OS/2
 * rules ...
 */
#define ProjectRoot      /XFree86

/* ... but not when a user tries to compile something
 * from a different drive Please report any directories
 * missed to Holger.Veit@gmd.de
 */
#ifdef UseInstalled
#define IncRoot $(X11ROOT)/XFree86/include
#define LdPreLib -L$(X11ROOT)/XFree86/lib
#endif

/* more directories */
#define AdmDir    $(LIBDIR)/adm
#define ConfigDir $(X11ROOT)$(LIBDIR)/config

```

```

        DESTDIR = $(X11ROOT)

/* for the man pages: Make clear they are from X11 */
#define BookFormatManPages YES
#define ManDirectoryRoot /XFree86/man/man
#define ManSuffix      1X11
#define LibManSuffix   3X11
#define FileManSuffix  4X11
#define XmanSearchPath /XFree86/man/man
#define ManSourcePath  $(MANPATH)

/* you must have installed groff! */
#define TblCmd      tbl
#define ColCmd      cat
#define NeqnCmd     eqn
/* groff -Tascii -mandoc is for format portability */
#define NroffCmd    groff -Tascii -mandoc
#define EqnCmd      eqn -Tascii
#define TroffCmd    groff -Tps
                NEQN = NeqnCmd
                NROFF = NroffCmd

/* A set of standard defines (warning: underscored emx
 * extensions are now excluded from the headers unless
 * -D_WITH_UNDERSCORE is defined):
 */
#define OSDefines -D__EMX__ -D__i386__ -DOS2_ \
-Demxos2 -D_POSIX_SOURCE -D_BSD_SOURCE -D_GNU_SOURCE

#ifndef BuildXKB
# define BuildXKB          YES
#endif
#ifndef BuildImplibs
# define BuildImplibs     YES
#endif
#ifndef BuildLBX
# define BuildLBX         YES
#endif

/* some more properties of the OS/2 implementation */
#define HasNdbm            YES
#define HasPutenv          YES
#define HasSockets        YES
#define HasSnprintf       YES
#define HasBsearch        NO
#define HasLargeTmp       NO
#define HasPoll            NO
#define HasBSD44Sockets   NO
#define HasSecureRPC      NO
#define HasSymLinks       NO

```

```

#define HasVFork                NO
#define HasVarDirectory         NO
/* #define HasStrcasecmp NO
   see below */
#define HasStrcasecmp           YES
#define Malloc0ReturnsNull      YES
#define SetTtyGroup             NO
#ifndef HasLibCrypt
# define HasLibCrypt            NO
#endif
#ifndef HasShm
# define HasShm                  YES
#endif
#define ConnectionFlags -DTCPCONN -DOS2PIPECONN
#define StdIncDir              $(C_INCLUDE_PATH)

/* XF86Setup stuff - does not work yet! */
#define HasTcl                  NO
/*#define TclLibDir $(X11ROOT)$(SHLIBDIR) */
/*#define TclIncDir $(X11ROOT)$(INCRROOT)/tcl */
/*#define TclLibName xtcl */

#define HasTk                    NO
/*#define TkLibDir $(X11ROOT)$(SHLIBDIR) */
/*#define TkIncDir $(X11ROOT)$(INCRROOT)/tk */
/*#define TkLibName xtk */

/*
 * Compiler Features
 */
#define HasGcc                    YES
#define HasGcc2                   YES
#define HasCplusplus              YES
#define HasGcc2ForCplusplus       YES
#define GccUsesGas                 YES
#define UseGas                      YES
#define GnuCxx                     YES
#define DoRanlibCmd                YES
#define NeedConstPrototypes        YES
#define NeedFunctionPrototypes     YES
#define NeedNestedPrototypes       YES
#define NeedVarargsPrototypes      YES
#define NeedWidePrototypes         NO

#define CxxCmd cxx
#define StandardCxxDefines         -traditional
#define PreProcessCmd              CxxCmd

#define CcCmd gcc -Wall -mpentium -O2
#define DefaultCCOptions -D__ST_MT_ERRNO__ \

```

```

-Dstrcasecmp=strcmp -Zmtd
#define LibraryCCOptions -D__ST_MT_ERRNO__ \
-Dstrcasecmp=strcmp -Zsysv-signals -Zmtd
#define ServerCCOptions -D__ST_MT_ERRNO__ \
-Dstrcasecmp=strcmp -Zsysv-signals -Zmtd

#define CplusplusCmd g++ -O3 -Wall -mpentium /* or gcc */
#define CplusplusOptions -Dstrcasecmp=strcmp -Zmtd \
-Zsysv-signals

#define MakeCmd x11make
#define AsCmd gcc -S
#define GccGasOption -DGCCUSESGAS
#define AsmDefines -DUSE_GAS

#define InstallCmd install

#define LdCmd ld
#define ExtraLoadFlags -Zbin-files -Zmtd -Zsysv-signals
#define ExtraLibraries -lsocket -lbsd

/* quite a number of programs you need */
#define ArCmd ar cq
#define RanlibCmd ar s
#define BourneShell /**/
#define LexCmd flex -l
#define LexLib -lfl
#define YaccCmd bison
#define LintCmd /**/
#define MvCmd mv
#define CompressCmd compress
#define GzipCmd gzip
#define LnCmd cp
#define CpCmd cp
#define RmCmd ImakeHelper 4

#ifndef UseInstalled
#define ImakeCmd \imake
#define MkdirHierCmd \mkdirhier
#define DependCmd \makedepend
#else
/* imake and mkdirhier come from Imake.tmpl, but DependCmd must be overridden */
#define DependCmd makedepend
#endif

/* "shell scripts" in OS/2 have this extension */
#define SHsuf cmd

#define InstPgmFlags /**/
#define InstBinFlags /**/

```

```

#define InstUidFlags /**/
#define InstLibFlags /**/
#define InstIncFlags /**/
#define InstManFlags /**/
#define InstDatFlags /**/
#define InstallFlags /**/

/*#define OptimizedCDebugFlags DefaultGcc2i3860pt*/
#define OptimizedCDebugFlags
#define ServerOSDefines XFree86ServerOSDefines -DDXTIME
#define ServerExtraDefines GccGasOption XFree86ServerDefines

#if HasShm
# define ServerExtraSysLibs      -lshm
#endif

/*
 * Make & install Features
 */

#define AvoidNullMakeCommand      YES
_NULLCMD_ = @ rem
#define NullMakeCommand @ rem
#define StripInstalledPrograms    NO
#define CompressAllFonts          YES
#define CompressManPages          YES
#define GzipFontCompression       YES
#define DefaultUserPath           ./os2;/emx/bin;/tcpip/bin;$(BINDIR)
#define DefaultSystemPath         /os2;/emx/bin;$(BINDIR)

#ifndef ExtraFilesToClean
# define ExtraFilesToClean *.~* *.exe *.dll *.obj *.lib \
    *.map
#endif

#if CompressManPages
#define CompressManCmd  gzip -n
    COMPRESSMANCMD = CompressManCmd
#endif

/* Use ProgramTargetName() macro: OS/2 has '.exe' suffix
 * Change your Imakefile!
 */
#define ProgramTargetName(target) target.exe

/* we don't name libraries lib*.a
 * Change your Imakefile!
 */
#define LibraryTargetName(libname) libname.a
#define LibraryTargetNameSuffix(libname,suffix) Concat(libname,suffix.a)

```

```

/* ... and we even don't do it in rules that should have
 * used the above LibraryTargetName() macro
 */
#ifndef UnSharedLibReferences
#define UnSharedLibReferences(varname,libname,libsource)          @@\
Concat3(DEP,varname,LIB) = _UseCat($(USRLIBDIR)/,$(BUILDLIBDIR)/,libname.a)  @@\
Concat(varname,LIB) = LoaderLibPrefix Concat(-l,libname)        @@\
LintLibReferences(varname,libname,libsource)
#endif

#ifndef SharedLibReferences
#define SharedLibReferences(varname,libname,libsource,revname,rev)  @@\
Concat3(DEP,varname,LIB) = SharedLibDependencies(libname,libsource,revname) @@\
Concat(varname,LIB) = LoaderLibPrefix Concat(-l,libname)        @@\
LintLibReferences(varname,libname,libsource)
#endif

#include <os2.rules>
#include <os2Lib.rules>

# include <xfree86.cf>

#else
#error Edit os2.cf for your (non-EMX) OS/2 system
#endif

```

Reasons for that: Never edit generated Makefiles by hand: Far too much work for larger projects and errors are to be expected. (But it is sometimes unavoidable). If things work and don't break other projects, they should be mailed to the respective lists in order to finally become standard.

7.3 Common initial problems

Initially I thought imake was a labyrinthic beast, deserving the title of the Minotaurus among the make tools. But with small and simple fixes to work around some annoying bugs I was able to tame it a bit, so that 'xmkmf' at least does something useful:

7.3.1 'xmkmf -a' crashes because of 'sh' and 'Rexx' conflicts

Usually Imakefiles silently assume they are running on UNIX in a Bourne 'sh' environment, since authors intentionally do not support non UNIX platforms.

Just try running the following changed Rexx scripts first, to create a working environment (`make.cmd`):

```

/* REXX */
'@echo off' PARSE ARG a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
/* x11make.exe 'MAKE=x11make.exe SHELL=d:/os2/cmd.exe' a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
'SET SHELL=D:OS2CMD.EXE'
'SET X11SHELL=D:OS2CMD.EXE'
'set emxshell=D:OS2CMD.EXE'
'set CONFIG_SHELL=D:OS2CMD.EXE'
'set MAKE_SHELL=D:OS2CMD.EXE'
x11make.exe 'MAKE=x11make.exe SHELL= MAKE_SHELL=\os2\cmd.exe
CONFIG_SHELL=d:os2cmd.exe' a1 a2 a3 a4 a5 a6 a7 a8 a9 a10

```

or `gmake.cmd`

```

/* REXX */
'@echo off'
PARSE ARG a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
'set SHELL=d:/bin/ash.exe'
'SET X11SHELL=D:OS2CMD.EXE'
'set emxshell=d:/bin/ash.exe'
'set MAKE_SHELL=d:/bin/ash.exe'
/* x11make.exe 'MAKE=x11make.exe SHELL=' a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 */
x11make.exe 'MAKE=x11make.exe SHELL=d:/bin/ash.exe MAKE_SHELL=d:/bin/ash.exe CONN

```

Normally, running `make.cmd` should be sufficient, since the current OS/2 implementation by H. Veit and others makes extensive use of Rexx, needs `cmd.exe` as command processor and is incompatible with Bourne shells. Here a number of problems arise with many common Imakefiles.

Now lets assume `xmkmf` worked. Theoretically the steps are:

```

xmkmf; make Makefiles; make depend; make all
make install; make install.man

```

7.3.2 'make' crashes because of 'sh' and 'Rexx' conflicts

Imake is misconfigured on most (commercial) UN*X platforms. Authors normally compensate this by including many lines of (normally highly complicated) Bourne shell code and calling several Bourne shell scripts in the Imakefile. In this case 'make' won't work. Nor does a `sh.exe` as make shell work: The current implementation depends on `cmd.exe` and Rexx. Finding and rewriting everything in those immense Imakefiles, helper scripts and Makefiles isn't really an alternative. Just try alternatively `make` and `gmake` (having installed e.g. `/bin/ash.exe`). Now `make` should simply resume where it has crashed. Just delete 'products' of those crashes (empty script output, truncated object files, core dumps, etc.) before resuming the process. This makes using Imake real fun!

Tips by John Williams <jwilliams@commerce.otago.ac.nz> (concerning 'configure' of GNU 'autoconf', fixed by the OS/2-port, but still valid for certain parallel Bourne sh-scripts very common in Imakefiles and/or `config.local`, etc.):

"3. Configure couldn't search the OS/2 path for programs because the shell construct:

```
for foo in bar do blah
```

apparently needs 'bar' to be a space or colon delimited list, whereas the OS/2 \$PATH is semi-colon delimited.

4. Configure could not check to see if it's test programs were being compiled because it tests for success using `test -s` which means test that the file is not empty. The emx port of GCC creates `foo.exe` executables and touches `foo`, leaving an empty file. [...]

5. Overcome the above problems by setting an environment variable `UNIX_PATH` to be a colon delimited path. Next, search and replace using `sed` thus:

```
sed -f os2patch.sed configure > configure.os2
```

where `os2patch.sed` looks like:

```
s/PATH/UNIX_PATH/ s/test -s/test -f/ s/host=NONE/host=OS2/ s/"${IFS}:"$/"${IFS};"/
```

and then execute `sh Configure.os2`. Lines three and four are due to Ilya Zakharevich, who also pointed out the following issues:

1. if `/tmp` doesn't exist, make it
2. ensure `CONFIG_SHELL=sh` is set in shell environment you are running `Configure` from.
6. You can find what functions are contained in what object files and libraries by using `nm` (comes with `emx`). This also tells you which functions are referenced (called) by functions in that file."

7.4 Imakefile examples

`config/README` states: "The easiest way to write an Imakefile is to find another one that does something similar and copy/modify it!"

Here some strongly shortened examples that use Bourne shell code: `sh -c 'for i in "$(TEXT) $(SRCS) $(MISC) $(HDRS)"; do... etc., etc.`

7.4.1 creating a static library (archive): `NormalLibraryTarget(rw, $(OBJS))`

Excerpt from Imakefile for `xpaint` read/write directory:

```

#include "../version"
#include "../Local.config"
INCLUDES = -I.. $(XPM_INCLUDE) $(TIFF_INCLUDE) \
$(JPEG_INCLUDE) $(PNG_INCLUDE)

# Nothing to change below here
TIFF_SRC = writeTIFF.c readTIFF.c
#ifdef HaveTIFF
TIFF_OBJ = writeTIFF.$O readTIFF.$O
TIFF_DEF = -DHAVE_TIFF
#elseif
[...]
XPM_SRC = readWriteXPM.c
XPM_OBJ = readWriteXPM.$O
SGI_SRC = readWriteSGI.c
#ifdef SGIArchitecture
SGI_OBJ = readWriteSGI.$O
SGI_DEF = -DHAVE_SGI
#elseif

DEFINES = $(XPM_DEF) $(TIFF_DEF) $(SGI_DEF) \
$(JPEG_DEF) $(PNG_DEF)

SRCS = rwTable.c readWriteXBM.c readWritePNM.c \ readWriteXWD.c writePS.c read
$(PNG_SRC) libpnmrw.c

OBJS = rwTable.$O readWriteXBM.$O readWritePNM.$O \
readWriteXWD.$O writePS.$O readGIF.$O writeGIF.$O \
$(XPM_OBJ) $(TIFF_OBJ) $(SGI_OBJ) $(JPEG_OBJ) \
$(PNG_OBJ) libpnmrw.$O

HDRS = libpnmrw.h rwTable.h
MISC = Imakefile
NormalLibraryTarget(rw, $(OBJS))

makelist: sh -c 'for i in "$$(TEXT) $(SRCS) $(MISC) $(HDRS)"; do echo $(CURRE

```

7.4.2 creating an executable: ComplexProgramTarget(xart)

```

#include "../version"
#include "../Local.config"
DEFINES = $(ARCH_DEFINES) $(EXTRA_DEFINES) \
$(XPM_INCLUDE) -DXPAINT_VERSION="$(VERSION)"

XPSRC = [...]

SRCS = $(XPSRC) $(OPSRC) $(XPWIDSRC)
OBJS = $(XPOBJ) $(OPOBJ) $(XPWIDOBJ)

```

```

HDRS = [...]

SYS_LIBRARIES = XawClientLibs -lm

#ifdef SGIArchitecture
SGI_LIB = -limage
#endif

DEPLIBS = always xart.man xart.ad
LOCAL_LIBRARIES = LibraryTargetName(rw/librw.a) $(XPM_LIB) $(TIFF_LIB)
$(SGI_LIB) $(JPEG_LIB) $(PNG_LIB)

SUBDIRS = rw

ComplexProgramTarget(xart)
InstallAppDefaults(XArt)

NamedMakeSubdirs(always, $(SUBDIRS))
MakefileSubdirs($(SUBDIRS) bitmaps Doc)
CleanSubdirs($(SUBDIRS))

# Other targets...

clean::
    rm -f xart.ad.h DefaultRC.txt.h Help.txt.h PGP.* \
        xart.man xart.ad

includes:: xart.ad.h DefaultRC.txt.h Help.txt.h

xart.man:      xart.man.in version
    sed -e 'sáXPAINT_VERSIONá$(VERSION)á' < xart.man.in > $@

Colormap.$0: ColormapP.h Colormap.h
[...]
makelist::
    -$(RM) filelist
    @touch filelist

makelist::
    sh -c 'for i in "$$(TEXT) $(SRCS) $(MISC) $(HDRS)";
    do echo $(CURRENT_DIR)/$$i >> $(TOP)/filelist ; done '

NamedTargetSubdirs(makelist, $(SUBDIRS) bitmaps Doc, , , makelist)

kit: makelist
    sh -c 'sum="'cat filelist'" ; makekit -oMANIFEST MANIFEST
    $$sum'

tar: makelist

```

```

sh -c 'cd .. ; rm -f xpaint-$(VERSION).tar.gz;
tar czf xpaint-$(VERSION).tar.gz
'sed -e "s:^./::" -e "s:^:xpaint/:" xpaint/filelist'

TAGS: $(SRCS) $(HDRS) $(RWSRC)
      etags -t -o ./TAGS $(XPSRC) $(OPSRC) $(XPWIDSRC) $(RWSRC)
$(HDRS)

certification: makelist
              -$(RM) $(CERTIFICATION)
              certify 'cat filelist'

```

7.5 Some maybe useful defines

You may want to add parts of it to your Imakefile (or Local.config, etc.) in case of problems. Use comments, else you'll forget about these hackish def's!! Better: Include the necessary defines in config.h. You find them below being commented out (XCOMM XCOMM).

Beware: Redefines of functions are debugging headaches: The actual code won't be explicitly in the sources; you will forget, that you redef'd! Awful coding style!! Read the Portability chapter.

Giant command lines may require a good shell with a large input buffer. You can avoid this by adding necessary redefinitions to 'config.h' and not to the command line (-Dfoo).

```

XCOMM XCOMM /* OS/2 does not have everything */
#ifdef OS2Architecture
XCOMM XCOMM remove 'XCOMM XCOMM' comments, if you receive linkage errors
XCOMM XCOMM -DNEED_STRCASECMP
#include <string.h>
#define StrcasecmpDefines -Dstrcasecmp=stricmp \
    -Dstrncasecmp=strnicmp -Dmemcasecmp=memicmp

ARCH_DEFINES = -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1
-DHAVE_PARAM_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA_H=1 -DHAVE_ALLOCA=1
-DHAVE_STRERROR=1 -Dlstat=stat -DS_ISLNK=false -DS_ISBLK=false
-Dsymlink()(=0) -Dreadlink(s,t,l)=(strcpy(t,s),strlen(t))

XCOMM XCOMM Try to include it in config.h
XCOMM XCOMM #define S_ISLNK() false
XCOMM XCOMM #define S_ISBLK() false
XCOMM XCOMM #define symlink() (0)
XCOMM XCOMM #define readlink(s,t,l) (strcpy(t,s),strlen(t))

XCOMM XCOMM /* Compare S1 and S2, ignoring case. */
XCOMM XCOMM int memicmp(__const__ char *, __const__ char *, size_t);

```

```

XCOMM XCOMM #ifndef memcasecmp
XCOMM XCOMM #define memcasecmp memicmp // emx uses M$ fn name
XCOMM XCOMM #endif

XCOMM XCOMM /* Compare S1 and S2, ignoring case. */
XCOMM XCOMM int stricmp(__const__ char *, __const__ char *);

XCOMM XCOMM #ifndef strcasecmp
XCOMM XCOMM #define strcasecmp stricmp
XCOMM XCOMM #define _strcasecmp stricmp
XCOMM XCOMM #endif

XCOMM XCOMM #if !defined(__STRICT_ANSI__) && !defined(_POSIX_SOURCE)

XCOMM XCOMM /* Compare no more than N chars of S1 and S2, ignoring case. */
XCOMM XCOMM int strnicmp(__const__ char *, __const__ char *, size_t);

XCOMM XCOMM #ifndef strncasecmp
XCOMM XCOMM #define strncasecmp strnicmp
XCOMM XCOMM #define _strncasecmp strnicmp
XCOMM XCOMM #endif

XCOMM XCOMM #endif /* !defined(__STRICT_ANSI__) */

XCOMM XCOMM /* drive letter support needs major rewriting:
XCOMM XCOMM * #define getcwd _getcwd2
XCOMM XCOMM * #define chdir _chdir2
XCOMM XCOMM */

XCOMM XCOMM /* Adapt hardcoded path names to XFree86/OS2 structure */

XCOMM XCOMM #if defined (__cplusplus)
XCOMM XCOMM extern "C" {
XCOMM XCOMM #endif
XCOMM XCOMM const char* __XOS2RedirRoot(const char* pathname);
XCOMM XCOMM #if defined (__cplusplus)
XCOMM XCOMM }
XCOMM XCOMM #endif

XCOMM XCOMM #define "/usr/X11/foo" (__XOS2RedirRoot("/xfree86/lib/x11/foo"))

#endif

```


Chapter 8

XFree86/2 and Multi-threading

Most \mathcal{X} applications are single-threaded, i.e. multi-threading effects are 'emulated' using the Unix `fork()` syscall.

The Port of XFree86 to OS/2 is not thread-safe, i.e. you can only make X11 calls from a single-thread. `emx select()` (including all \mathcal{X} i/o that depends on `select()`) are not (!) thread-safe. This means that using multi-threading within \mathcal{X} applications is for experts only. Calls to a second thread that interacts with \mathcal{X} will crash your application. (unless you use some inter-thread communication to serialize the calls).

Compile and link with

```
-D__ST_MT_ERRNO__ -Zmtd
```

This takes care of redefining ANSI '`errno`' for multithreading (cf. `emx docs`) and prepares for linkage with reentrant versions of the Lib C functions. Note, this is only a necessary and not a sufficient pre-condition for thread-safe programs.

Notes:

Existing servers and clients which use Posix threads (PTHREADS) have not yet been ported to OS/2. There is ongoing work to achieve PTHREADS compatibility for OS/2-emx, though. XFree86/2 libraries (DLL's) and applications are prepared for later extensive use of multi-threading.

Cf. `emx docs`:

"Use `_beginthread()` to start a new thread. Do not use `DosCreateThread` unless the new thread doesn't call C library functions.

The C library functions in `mt/c.a`, `mt/c.lib`, and `emxlibcm.dll` are not yet completely thread-safe. [...]

If you are very careful, you can write multithread programs that use the single-thread libraries. Only the main thread is allowed to call library functions that have side effects, including functions that set `errno`. All other threads should use OS/2 API functions instead. "[Parallel precautions apply to multithreaded \mathcal{X} applications.]

Chapter 9

Numerical Math

9.1 Types, Limits, Conversion.

For the following, please cf. the macros in `<limits.h>`:

The smallest integral type in C is named `char`, with the variants `signed char` and `unsigned char`. `char` differs from the other integral types in that it is not defined whether signed or unsigned. The next integral types are `short int` (often just referred to as `short`), `int` (including `enum !`) and `long int` (often just `long`.) All of them have unsigned versions.

To be on the safe side you must specify which you mean. However, conversions to other types require some precaution (see below). The maximum value for a char is `CHAR_MAX`, and the minimum `CHAR_MIN`, for signed char `SCHAR_MAX` and `SCHAR_MIN` and the maximum `unsigned char` is `UCHAR_MAX` (the minimum unsigned anything is always 0.)

There are three floating point types in C, `float`, `double` and `long double`. The floating point types are always signed. If you have operations with mixed types, the type with the largest value space decides the precision the operation will be done with. For `char` they're done at `int` precision. For `float` the arithmetics are done at `double` precision.

Integral Types: The value space of an integral type is not rigorously defined; check `CHAR_BIT`, which on OS/2 is always 8. ANSI C only requires that the value space of `short` is a subset of that of `int` which is a subset of that of `long`, and that `int` is the type most natural for the processor. For 32-bit OS/2 this means that `int` and `long` is 32 bits. The Alpha uses 16-bits for `short` and 64 bit `long`. Signed integral types are usually not quite symmetric due to the representation used (although they actually can be, this depends on the processor.) Usually the negative range is one number larger than the positive range. For example the maximum signed 16-bit number is 32767 and the minimum signed 16-bit number is -32768 for Intel processors.

Floats and Doubles : The smallest floating point type `float` is, just like the smallest integral type `char` a special case. Arithmetic is not done with them; they are just a way of storing data with a certain precision. Arithmetic is instead done with the precision of a larger type. K&R encouraged the interchangeable use of `float` and `double` since all expressions with such data types were always evaluated using the `double` representation - a real nightmare for those implementing efficient numerical algorithms in C.

According to the ANSI standard such programs will continue to exhibit the same behavior *as long as one does not prototype the function*. Therefore, when prototyping functions make sure the prototype is included when the function definition is compiled so the compiler can check if the arguments match.

- Keep in mind that the `double` representation does not necessarily increase the *precision*. Actually, in most implementations the worst-case precision decreases but the *range* increases.
- Do not use `double` unnecessarily since in most cases there is a large performance penalty. Furthermore, there is no point in using higher precision if the additional bits which will be computed are garbage anyway. The precision one needs depends mostly on the precision of the input data and the numerical method used.

Conversions are, as far as possible, sign preserving. Overflow wraps around, according to the processor model. With VisualAge C++, where `char` is unsigned by default, the output of the following becomes:

```
unsigned => 255, signed => -1, unknown => 255.
```

Emx gives:

```
unsigned => 255, signed => -1, unknown => -1:
```

```
#include <stdio.h>
int main(void) {
    unsigned char uc=255;
    signed char sc = 255;
    char c = 255;
    int iuc = uc, isc = sc, ic = c;
    printf(
"unsigned => %d, signed => %d, unknown => %d\n",
        iuc, isc, ic);
    return 0;
}
```

9.2 IEEE Maths.

Emx provides programming interfaces which comply to the recent standards of the ANSI mathematical amendments and IEEE, including infinity and NaN (not-a-number) values.

Infinity is a value with an associated *sign* that is mathematically *greater in magnitude than any binary floating-point number*.

NaN is a value in floating-point computations that is *not* interpreted as a *mathematical value*, and that contains a mask state and a sequence of binary digits.

Cf. `math.h` (and `gnumath.h` below) for the following:

The value of infinity can be computed from `1.0/0.0`. The value of a NaN can be computed from `0.0/0.0`. Depending on its bit pattern, a NaN can be either quiet (NaNQ) or signaling (NaNS), as defined in the ANSI/IEEE Standard for Binary Floating-Point Arithmetic (754-1982). A NaNQ is masked and never generates exceptions. A NaNS may be masked and may generate an exception, but does not necessarily do so. Emx currently supports only quiet NaN values; all NaN values discussed below refer to quiet NaNs.

NaN and infinity values are defined as macro constants in the `<math.h>` header file. You can get the corresponding `float` and `long double` values by a *cast*, you may obtain *negative* values by using the unary minus operator. The value of `0.0` can also be positive or negative, e.g. `1.0/(-0.0)` results in `-INFINITY`.

Note: Although positive and negative infinities are specific bit patterns, NaNs are not. A NaN value is not equal to itself or to any other value. For example, if you assign a NaN value to a variable `x`, you cannot check the value of `x` with the statement `if (NaN == x)`. Instead, use the statement `if (x != x)`. All relational and equality expressions involving NaN values always evaluate to `FALSE` or zero (0), with the exception of not equal (`!=`), which always evaluates to `TRUE` or one (1).

Software written for Unix on the other hand often relies on not generally portable vendor extensions. Often rewriting this proprietary code to ANSI/IEEE is not so difficult. One problem is obtaining machine constants. Sun's, DEC's, etc. include `<values.h>`. Emx follows the ANSI/IEEE Standard standard that such constants be defined in the header file `<float.h>`. Common problems are (cf. T. Love, ANSI C, `<tpl@eng.cam.ac.uk>`):

Testing for equality: Real numbers are handled in ways that don't guarantee expressions to yield exact results. It's risky to test for equality. Better is to use something like

```
d = max(1.0, fabs(a), fabs(b))
```

and then test `fabs(a - b)/d` against a relative error margin.

Useful constants in `float.h` are `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON`, defined to be the smallest numbers such that

```
1.0f + FLT_EPSILON != 1.0f
1.0  + DBL_EPSILON != 1.0
1.0L + LDBL_EPSILON != 1.0L
```

respectively.

Avoiding over- and underflow: You can test the operands before performing an operation in order to check whether the operation would work. You should always avoid dividing by zero. For other checks, split up the numbers into fractional and exponent part using the `frexp()` and `ldexp()` library functions and compare the resulting values against `HUGE_VAL` (all in `<math.h>`).

Infinity:

The IEEE standard for floating-point recommends a set of functions to be made available. Among these are functions to classify a value as `NaN`, `Infinity`, `Zero`, `Denormalized`, `Normalized`, and so on. Most implementations provide this functionality, although there are no standard names for the functions. Such implementations often provide predefined identifiers (such as `_NaN`, `_INFINITY`, etc) to allow you to generate these values.

If `x` is a floating point variable, then `(x != x)` will be `TRUE` if and only if `x` has the value `NaN`. Many Unix C implementations claim to be IEEE 754 conformant, but if you try the `(x!=x)` test above with `x` being a `NaN`, you'll find that they aren't.

In the mean time, you can write your own 'standard' functions and macros, and provide versions of them for each system you use. If the system provides the functions you need, you `#define` your 'standard' functions to be the system functions. Otherwise, you write your function as an interface to what the system provides, or write your own from scratch.

9.3 Useful Math Extensions

Some useful optimized extensions (functions and macros) are declared and defined in this header (simply include it instead of `math.h`):

```
/* gnumath.h (emx+gcc)
   Declarations and definitions for additional math functions.
   */
/*
   Parts: Cephes Math Library Release 2.3:  March, 1995
         Copyright 1984, 1995 by Stephen L. Moshier
   Documentation is included on the distribution media as
   Unix-style manual pages that describe the functions and their
   invocation.  The primary documentation for the library functions
   is the book by
         Moshier, Methods and Programs for Mathematical Functions,
         Prentice-Hall, 1989.
   Parts: Copyright (C) 1991, 92, 93, 95, 96, 97, 98 Free Software
   Foundation, Inc. This file is part of the GNU C Library.
   See the GNU Library General Public License for more details.
   *
   *   Extensions to ISO C Standard: 4.5 MATHEMATICS   <math.h>
```

```

*/
#ifndef _GNUMATH_H__
#define _GNUMATH_H__

#if ( !defined (__STRICT_ANSI__) && !defined (_POSIX_SOURCE) && \
!defined (__NO_C9X) ) || defined (_WITH_UNDERSCORE)
#undef _WITH_UNDERSCORE
#define _WITH_UNDERSCORE 1

/* Machine-dependent inline versions */
#if (!__GNUC__ || __GNUC__ < 2 || \
(__GNUC__ == 2 && __GNUC_MINOR__ <= 7))
/* The gcc, version 2.7 or below, has problems with all this inlining
code. So disable it for this version of the compiler. */
# undef __NO_MATH_INLINES
# define __NO_MATH_INLINES
#else
# define MATH_INL static __inline__
MATH_INL double expm1(double);
MATH_INL double log1p(double);
MATH_INL long double log1pl(long double);
MATH_INL double asinh(double);
MATH_INL long double asinhl(long double);
MATH_INL double acosh(double);
MATH_INL long double acoshl(long double);
MATH_INL double atanh(double);
MATH_INL long double atanh1(long double);
MATH_INL double cosh1(double);
MATH_INL double acosh1p(double);
MATH_INL double logb(double);
MATH_INL double drem(double, double);
MATH_INL void sincos(double, double *, double *);
MATH_INL double pow2(double);
MATH_INL double hypot(double x, double y);
#endif /* __NO_MATH_INLINES */
#endif /* ANSI */

/* Get machine-dependent HUGE_VAL value (returned on overflow).
On all IEEE754 machines, this is +Infinity.
SVID wants HUGE instead of infinity.
Get machine-dependent NAN value (returned for some domain errors). */
#include <math.h>

/* X/Open wants another strange constant. */
#include <ieee.h> /* MAXFLOAT */
#include <float.h> /* Some useful constants. */
#include <limits.h>

#if !defined (__STRICT_ANSI__) && !defined (_POSIX_SOURCE) && \
!defined (__NO_C9X)

```

```

/* The above constants are not adequate for computation using 'long double's.
   Therefore we provide as an extension constants with similar names as a
   GNU extension. (emx gcc version)
*/
#define M_E1          2.71828182845904523536L      /* e */
#define M_LOG2E1     1.44269504088896340736L      /* log 2e */
#define M_LOG10E1    0.434294481903251827651L     /* log 10e */
#define M_LN21       0.693147180559945309422L     /* log e2 */
#define M_LN101     2.302585092994045684018L     /* log e10 */

#define M_PI1        3.14159265358979323846L
#define M_PI_21     1.57079632679489661923L      /* pi/2 */
#define M_PI_41     0.785398163397448309616L      /* pi/4 */
#define M_1_PI1     0.318309886183790671538L      /* 1/pi */
#define M_2_PI1     0.636619772367581343076L      /* 2/pi */
#define M_2_SQRTPI1 1.12837916709551257390L      /* 2/sqrt(pi) */
#define M_SQRT21    1.41421356237309504880L      /* sqrt(2) */
#define M_SQRT1_21  0.707106781186547524401L      /* 1/sqrt(2) */

/* Some Useful Constants From Cephes Math Library */
#define MAXNUML      LDBL_MAX
#define MACHEPL      LDBL_EPSILON
#define MAXLOGL      1.1356523406294143949700E+04L /* log1(MAXNUML) */
#define MINLOGL      (-1.1355137111933024058900E+04L) /* log1(LDBL_MIN) */
#define LOGE2        6.9314718055994530941723E-1
#define LOGE2L       6.9314718055994530941723E-1L
#define LOG2EL       1.4426950408889634073599E0L
#define PI1          3.1415926535897932384626L
#define PIO2L        1.5707963267948966192313L
#define PIO4L        0.78539816339744830961566E-1L
#define NEGZEROL     (-0.0L)
#define NANL          (long double)NAN
#define INFINITYL    (long double)INFINITY
#define NANF          (float)NAN
#define INFINITYF    (float)INFINITY
#include <errno.h>

#if !defined __NO_MATH_INLINES || defined _MATH_INLINES

#define _sgn(x) (x == 0.0 ? 0.0 : (x > 0.0 ? 1.0 : -1.0))
#define _sgnl(x) (x == 0.0L ? 0.0L : (x > 0.0L ? 1.0L : -1.0L))
#define _sgn1(x) (x >= 0.0 ? 1.0 : -1.0)
#define _sgn1l(x) (x >= 0.0L ? 1.0L : -1.0L)

/* BSD Functions */
MATH_INL double
drem(double x, double y) {
    register double value;
    __asm__ __volatile__ ("1: fprem1\n\t" "fstsw %%ax\n\t" "sahf\n\t" "jp 1b"

```

```

        : "=t" (value) : "0" (x), "u" (y) : "ax", "cc");
    return value;
}

    MATH_INL double
expm1(double x) {
    register double value, exponent, temp;
    __asm__ __volatile__
        ("fldl2e                # e^x - 1 = 2^(x * log2(e)) - 1\n\t"
         "fmul    %%st(1)        # x * log2(e)\n\t"
         "fstl    %%st(1)\n\t"
         "frndint                # int(x * log2(e))\n\t"
         "fxch\n\t"
         "fsub    %%st(1)        # fract(x * log2(e))\n\t"
         "f2xm1                # 2^(fract(x * log2(e))) - 1\n\t"
         "fscale                # 2^(x * log2(e)) - 2^(int(x * log2(e)))\n\t"
         : "=t" (value), "=u" (exponent) : "0" (x));
    __asm__ __volatile__
        ("fscale                # 2^int(x * log2(e))\n\t"
         : "=t" (temp) : "0" (1.0), "u" (exponent));
    temp -= 1.0;
    return temp + value;
}

/* Optimized versions for some non-standardized functions. */
    MATH_INL double
hypot(double x, double y) {
    return sqrt(x * x + y * y);
}

    MATH_INL double
log1p(double x) {
    register double value;
    if (fabs (x) >= 1.0 - 0.5 * M_SQRT2)
        value = log(1.0 + x);
    else
        __asm__ __volatile__
            ("fldln2\n\t"
             "fxch\n\t"
             "fyl2xp1"
             : "=t" (value) : "0" (x));
    return value;
}

    MATH_INL long double
log1pl(long double x){
    register long double value;
    if (fabsl(x) >= 1.0L - 0.5L * M_SQRT21)
        value = logl(1.0 + x);
    else
        __asm__ __volatile__
            ("fldln2\n\t"
             "fxch\n\t"

```

```

        "fyl2xp1"
        : "=t" (value) : "0" (x));
    return value;
}

    MATH_INL double
asinh(double x) {
    register __const__ double y = fabs(x);
    if (__isnan(x)) return x;
    if (x == 0.0) return x;
    if (!(__isfinite(x))) return x;
    if (y > 1.0e10) return log(x) + LOGE2;
#ifdef GNUFNS /* This is the GNU original and it is wrong!!! */
/*    return log1p ((y * y / (sqrt (y * y + 1.0) + 1.0) + y)
        * _sgn1(x));
*/
    return log1p(y +(y * y/(sqrt(y * y + 1.0)+ 1.0)))* _sgn1(x);
#else
    return  log(x +      sqrt(x * x + 1.0)      );
#endif
}

    MATH_INL double
acosh(double x) {
    if (__isnan(x)) return x;
    if (x < 1.0) {errno = EDOM; return NAN;}
    if (x > 1.0e10) {
        if (x == INFINITY) return INFINITY;
        return(log(x) + LOGE2); }
    return log(x + sqrt(x - 1.0) * sqrt(x + 1.0));
}

    MATH_INL double
atanh(double x) {
    register __const__ double y = fabs(x);
    if (x == 0.0) return x;
    if (y >= 1.0) {
        if (x == 1.0) return INFINITY;
        if (x == -1.0) return -INFINITY;
        errno = EDOM; return NAN; }
    if (y < 1.0e-8) return x;
#ifdef GNUFNS
    return -0.5 * log1p(-(y + y)/(1.0 + y)) * _sgn1(x);
#else
    return 0.5 * log ( (1.0 + x)/(1.0 - x) );
#endif
}

    MATH_INL long double
asinh1(long double x) {
    register __const__ long double y = fabs1(x);
    if (__isnanl(x)) return x;
    if (x == 0.0L) return x;
    if (!(__isfinitel(x))) return x;

```

```

        if (y > 1.0e10L) return logl(x) + LOGE2L;
#ifdef GNUFNS
        return log1pl( (y * y / (sqrtl(y * y + 1.0L) + 1.0L) + y) )
            * _sgn1l(x);    /* This is corrected */
#else
        return logl( x + sqrtl(x * x + 1.0L));
#endif
}

MATH_INL long double
acoshl(long double x) {
    if (__isnanl(x)) return x;
    if (x < 1.0L) {errno = EDOM; return NANL;}
    if (x > 1.0e10L) {
        if (x == INFINITYL) return INFINITYL;
        return logl(x) + LOGE2L; }
    return logl(x + sqrtl(x - 1.0L) * sqrtl(x + 1.0L));
}

MATH_INL long double
atanhl(long double x) {
    register __const__ long double y = fabsl(x);
    if (x == 0.0L) return x;
    if (y >= 1.0L) {
        if (x == 1.0L) return INFINITYL;
        if (x == -1.0L) return -INFINITYL;
        errno = EDOM; return NANL; }
    if (y < 1.0e-8L) return x;
#ifdef GNUFNS
    return -0.5L * log1pl(-(y + y)/(1.0L + y)) * _sgn1(x);
#else
    return 0.5L * logl( (1.0L + x)/(1.0L - x) );
#endif
}

MATH_INL double
coshm1(double x) {
    register __const__ double __exm1 = expm1 (fabs (x));
    return 0.5 * (__exm1 / (__exm1 + 1.0)) * __exm1;
}

MATH_INL double
acosh1p(double x) {
    return log1p(x + sqrt (x) * sqrt (x + 2.0));
}

MATH_INL double
logb(double x) {
    register double value, __junk;
    __asm__ __volatile__
        ("fextract\n\t"
         : "=t" (__junk), "=u" (value) : "0" (x));
    return value;
}

MATH_INL void
```

```

sincos(double x, double *__sinx, double *__cosx) {
  register double __cosr, __sinr;
  __asm__ __volatile__ ("fsincos"
    : "=t" (__cosr), "=u" (__sinr) : "0" (x));
  *__sinx = __sinr;
  *__cosx = __cosr;
}

MATH_INL double
pow2(double x) {
  register double value, exponent;
  int __p = (int) x;

  if (x == (double) __p)
    return ldexp (1.0, __p);

  __asm__ __volatile__
    ("fldl      %%st(0)\n\t"
     "frndint                    # int(x)\n\t"
     "fxch\n\t"
     "fsub      %%st(1)          # fract(x)\n\t"
     "f2xm1                    # 2^(fract(x)) - 1\n\t"
     : "=t" (value), "=u" (exponent) : "0" (x));
  value += 1.0;
  __asm__ __volatile__
    ("fscale"
     : "=t" (value) : "0" (value), "u" (exponent));

  return value;
}

#endif /* _POSIX_SOURCE */
#endif /* __NO_MATH_INLINES */

#endif /* GNUMATH_H__ */

```

Chapter 10

What Shall I Better Change in the Sources?

Last but not least some well meant recommendations from Chapter 5 of the "emx APPLICATION DEVELOPER'S GUIDE", with some additional comments. You might use them as a short checklist.

10.1 Globbing

" If you want Unix-like wildcard expansion built into the program, use

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

This should be done at the very beginning of main(), before ARGV and ARGV are used. See _wildcard() and _response()."

10.2 DOS "text" mode, binary reading, etc.

"Famous problems are related to the binary/text mode mess on OS/2, especially when using stdin/stdout." (Alexander Mai)

" Programs using stdin, stdout or stderr for binary data should call _fsetmode() to switch the stream to binary mode."

" Change all open(), fopen(), fdopen() and freopen() calls to use O_BINARY or "b", respectively, for binary files. If a file contains both binary and textual data, read the file in binary mode and do the conversion yourself."

Note: XFree86/OS2 uses '-Zbin-files'. So "do the conversion yourself" "and ignore carriage returns" and <CTRL Z>.

" 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."

10.3 Process management and `fork()`

"Replace `fork()` and `exec*()` with `spawn*()`. Under OS/2, `fork()` is inefficient."
(emx docs)

Note: This is usually not difficult; it is necessary to use '-Zomf'. Cf. the example below.

Replacing `vfork()`-type lightweight processes with multi-threading is complicated.

10.3.1 What is the difference between `fork()` and `spawnvp()`?

10.3.1.1 `fork()` and emx

The Posix syscall `fork()` is for traditional Unix systems the only possibility to create a new process, i.e. a nearly identical clone of an existing one:

"This is achieved by copying the pointers in the PROCESS INFORMATION BLOCK (PIB) of the parent to the newly allocated block in the kernel. On the other hand, on DOS (Windows included) and on OS/2 (Windows NT included) there is no "`fork`" system call but there is a "`spawn`" instead. This system call just allocates a *new* process information block. Only file *handles* and the environment *variable space* are inherited and selectors to the *address space* of the parent process are *not* copied. *So the child cannot access the contents of the memory of the parent.* In fact, for OS/2 and Windows NT, it is impossible to implement "`fork`" in the same way as on the UNIX, since even root has only read access to the PIB table in the kernel. [In fact, there is no such security leak as '`root`' in OS/2 (AH)]

The emx library function `fork()` is actually `spawn()` system call with a little bit of IPC to copy memory contents. It is inefficient because all the memory contents rather than just pointers are copied.

Well, the difference I was talking was that a *spawned* child has a clean address space, while a *forked* one might have a corrupted one." (SMiyata).

10.3.1.2 making it work

"fork() works only in programs linked by ld. It does not work in programs linked by LINK386." [-Zomf]

"fork() doesn't work correctly in multithread programs."

"If the parent process uses the general terminal interface for the keyboard, the child process cannot read from the keyboard using `_read_kbd()`, the general terminal interface, or the KBD OS/2 API functions.

If the process has a non-contiguous heap (that is, multiple heap objects), `fork()` will fail. Increase the initial heap size to work around this problem. If any DLL used by the program shares the heap with the program and uses the heap in `_DLL_InitTerm()` by calling `sbrk()`, `malloc()`, etc., `fork()` will fail. In both cases, `fork()` will return -1 and set `errno` to `ENOMEM`.

See also: `alarm()`, `exec*()`, `_rmutex_create()`, `sbrk()`, `sigaction()`, `signal()`, `spawn*()`, `_uflags()`, `wait()`, `waitpid()`"

See the `emx` reference for more details about caveats. Use huge stack and heap size.

10.3.2 replacing fork() and execvp(), etc.

In Posix the parent process clones itself with `fork()`, which generates a new process environment, whereas `exec()` overwrites its own present environment with a new one, keeping the PID. Usually this is combined: `exec()` is used in the forked child process, overwriting the parent's process status with a new one. Meanwhile the parent usually calls `wait()` or a home-brew function and waits for child termination, communicated by `SIGCHLD`.

`ps (pstat.exe)` gives you the details.

A common problem here is, that the child after termination does not find this `wait()` etc. point of the parent. Thus a 'zombie' remains. This may lead to a process table overflow, so that the system has to be shutdown and rebooted. All this hassle can be simplified by using `spawn*()`.

Excerpt from `exec*()` documentation:

" When the new process ends, the parent process will be notified by `SIGCHLD` and `wait()` will return the original process ID (PID). [...]

Restrictions:

Native DOS programs cannot be run. The new process gets a new process ID. Unless the new process is an `emx` program (using `emx.dll` or `emx.exe`), the actions for all signals are reset to `SIG_DFL` in the new process"

"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 SIGCLD). 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."

But for tracing and debugging:

"Do not use the `PTRACE_TRACEME` request of `ptrace()`: use `P_DEBUG` instead when starting the process with `spawn*()`."

10.3.3 `fork()/exec()` and dual way pipe communication

It's no simple replacement when porting UNIX `fork()` and `exec*()` pairs to OS/2's `spawn*()`. The following two programs can be used as an example:

The first using `fork()` and `execv()` should be linked with `ld` but I have no `*.a` libraries at hand so cannot test it. But I think it is correct.

The second is linked with `LINK386` and works fine. Both programs create a child process and then write a message through a pipe to the standard input of the child process. You can test it with `cat.exe`, which displays what has been read from `stdin`. This should give you a hint how to port `fork()` and `exec*()`. Look at the source of `BASH 2.0` for OS/2 for another way to tackle this problem.

Vincent Kuo

```

/* pipe1.c, using fork() and exec*() */
#include <unistd.h>
#include <io.h>
static const char message[] = "Hello, kid!";

int
main(int argc, char**argv)
{ /* use the name of the program as argument */
  int two_fd[2], two_other_fd[2], pid;
  if (pipe(two_fd)) /* the reading end is [0] */
    exit(1); /* error creating pipe ;-( */
  pid = fork(); /* now I am two! */
  if (pid) { /* we are the parent! */
    if (pid == -1)
      exit(2); /* error fork()ing! ;-( */
    else {
      close(two_fd[0]); /* close unneeded reading end */
      write(two_fd[1], message, sizeof(message));
      close(two_fd[1]);
    }
  }
  } else { /* we are the forked child */
    dup2(two_fd[0], STDIN_FILENO); /* connect reading
      end to the new process' stdin, 'close' stdin */

```

```

    /* dup2(two_other_fd[0], STDOUT_FILENO); */
    close(two_fd[0]); /* close the old file descriptor */
    close(two_fd[1]); /* close unneeded one */
    /* close(two_other_fd[0]); close(two_other_fd[1]); */
    execv("cat", NULL);
    exit(3);          /* if exec*() returns: Error ;-( */
}
return 0;
}
-----
/* pipe2.c, using spawn*() */
#include <process.h>
#include <fcntl.h>
#include <io.h>

static const char message[] = "Hello, kid!";

/* supply the name of the program you want to run as argument */
int
main(int argc, char**argv)
{
    int fd[2], ftmp;
    if (pipe(fd))
        exit(1); /* error creating pipe */
    ftmp = dup(0); /* save the stdin handle */
    dup2(fd[0],0); /* assign the read end of the pipe to stdin */ close(fd[0]);
    fcntl(ftmp, F_SETFD, FD_CLOEXEC);
    fcntl(fd[1], F_SETFD, FD_CLOEXEC);          spawnv(P_NOWAIT,argv[1],argv+1);
    /* restore the original stdin file, necessary if
       more input from stdin is needed */
    dup2(ftmp,0);
    close(ftmp);
    write(fd[1], message, sizeof(message));
    close(fd[1]);
    return(0);
}
-----
/*
 * Routine: popenRW
 * Returns: PID of child process
 * Action : Exec program connected via pipe, connect int fd's to
 *          both the stdin and stdout of the process.
 * Params : Command - Program to run
 *          Pipes   - Array of 2 ints to store the pipe descriptors
 *                  Pipe[0] writes to child's stdin,
 *                  Pipe[1] reads from child's stdout/stderr
 */
#include <process.h>
#include <fcntl.h>
#include <io.h>

```

```

#define INCLUDE_DOS
#include <os2.h>
int
popenRW(char *const argv[], int *pipes)
{
    HFILE Out1, Out2, In1, In2;
    HFILE Old0 = -1, Old1 = -1, Old2 = -1, Tmp;
    int pid;
    if (DosCreatePipe(&Out2, &Out1, 4096))
        return FALSE;
    if (DosCreatePipe(&In1, &In2, 4096))
    {
        DosClose (Out1);
        DosClose (Out2);
        return FALSE;
    }
    /* Save std{in,out,err} */
    DosDupHandle (STDIN, &Old0);
    DosSetFHState (Old1, OPEN_FLAGS_NOINHERIT);
    DosDupHandle (STDOUT, &Old1);
    DosSetFHState (Old2, OPEN_FLAGS_NOINHERIT);
    DosDupHandle (STDERR, &Old2);
    DosSetFHState (Old2, OPEN_FLAGS_NOINHERIT);
    /* Redirect std{in,out,err} */
    Tmp = STDIN;
    DosDupHandle (In1, &Tmp);
    Tmp = STDOUT;
    DosDupHandle (Out1, &Tmp);
    Tmp = STDERR;
    DosDupHandle (Out1, &Tmp);
    /* Close file handles not needed in child */

    DosClose (In1);
    DosClose (Out1);
    DosSetFHState (In2, OPEN_FLAGS_NOINHERIT);
    DosSetFHState (Out2, OPEN_FLAGS_NOINHERIT);
    /* Spawn we now our hoary brood. */
    pid = spawnvp (P_NOWAIT, argv[0], argv);
    /* Restore std{in,out,err} */
    Tmp = STDIN;
    DosDupHandle (Old0, &Tmp);
    DosClose (Old0);
    Tmp = STDOUT;
    DosDupHandle (Old1, &Tmp);
    DosClose (Old1);
    Tmp = STDERR;
    DosDupHandle (Old2, &Tmp);
    DosClose (Old2);
    if(pid < 0)
        {

```

```

        DosClose (In2);
        DosClose (Out2);
        return -1;
    }

    pipes[0] = In2;
    _setmode (pipes[0], O_BINARY);
    pipes[1] = Out2;
    _setmode (pipes[1], O_BINARY);
    /* Save ID of write-to-child pipe for pclose() */
    ll_insert ((LL_KEY) In2, (LL_VAL) pid);

    return pid;
}
/*
 * Routine: pclose
 * Returns: TRUE on success
 * Action : Close a pipe opened with popen();
 * Params : Pipe - pipe to close
 */
int
pclose (FILE *Pipe)
{
    RESULTCODES rc;
    PID pid, pid1;
    int Handle = fileno (Pipe);
    fclose (Pipe);
    rc.codeTerminate = -1;
    pid1 = (PID) ll_lookup ((LL_KEY) Handle);
    /* if pid1 is zero, something's seriously wrong */
    if (pid1 != 0)
    {
        DosWaitChild (DCWA_PROCESSTREE, DCWW_WAIT, &rc, &pid, pid1);
        ll_delete ((LL_KEY) Handle);
    }
    return rc.codeTerminate == 0 ? rc.codeResult : -1;
}
#if DIAGNOSTIC
void
main ()
{
    FILE *fp1, *fp2, *fp3;
    int c;
    ll_print ();
    fp1 = popen ("gcc --version", "r");
    ll_print ();
    fp2 = popen ("link386 /?", "r");
    ll_print ();
    fp3 = popen ("dir", "r");
    ll_print ();
}

```

```

while ((c = getc (fp1)) != EOF)
    printf ("%c", c);
while ((c = getc (fp2)) != EOF)
    printf ("%c", c);
while ((c = getc (fp3)) != EOF)
    printf ("%c", c);
pclose (fp1);
ll_print ();
pclose (fp2);
ll_print ();
pclose (fp3);
ll_print ();
return;
}
#endif /* DIAGNOSTIC */

```

10.3.4 Example (C++-Code from modified LyX 1.0.4 sources):

The following example demonstrates how the `emx spawnvp()` interface brings about a considerable simplification of the initial `fork()`, `execvp()`, `waitForChild()` implementation. The frequent usage of `'#ifdef __EMX__'` reduces readability, though.

interface header

```

// -*- C++ -*-
#include <sys/types.h>
#include <LString.h>

#ifdef __GNUG__
#pragma interface
#endif

/// Class to controll system-calls according to OS-specific interface
/**@Doc:
Class, which controls a system-call according to the specific interface of
your operating system.

Instance starts and represents child processes.
You should use this class if you need to start an external program in LyX.
You can start a child in the background and have a callback function
executed when the child finishes by using the DontWait starttype.
*/
class Systemcalls {
public:
    ///
    enum Starttype {
        System,

```

```

        Wait,
        DontWait
};

// Callback function gets commandline and returnvalue from child
typedef void (*Callbackfct)(LString cmd, int retval);

//
Systemcalls();

/** Generate instance and start childprocess
    The string "what" contains a commandline with arguments separated
    by spaces.
    When the requested program finishes, the callback-function is
    called with the commandline and the returnvalue from the program.
    The instance is automatically added to a timercheck if starttype is
    DontWait (i.e. background execution). When a background child
    finishes, the timercheck will automatically call the callback
    function.
    */
Systemcalls(Starttype how, LString what, Callbackfct call = 0);

//
~Systemcalls();

/** Start childprocess. what contains a command on systemlevel.
    */
int Startscript(Starttype how, LString what, Callbackfct call = 0); // for reuse

/** gets PID of childprocess. Used by timer */
inline pid_t Getpid() { return pid; }

// Start callback
inline void Callback() { if (cbk) cbk(command, retval); }

/** Set return value. Used by timer */
inline void setRetVal(int r) { retval = r; }
private:
// Type of execution: system, wait for child or background
Starttype    start;
// Callback function
Callbackfct  cbk;
// Command line
LString      command;
// Process ID of child
pid_t        pid;
// Return value from child
int          retval;

```

```

        ///
        int Startscript();

        /// Spawn child according to the interface of your OS (AHanses)
        pid_t Fork(int spawnFlag);

        /// Wait for child process to finish. Updates returncode from child.
#ifdef __EMX__
        /**
        Waitforchild not necessary with spawnvp() of emx.
        Note (concerning OS/2 specifics): Emx interface implementation of
        spawnvp(P_WAIT, ...) returns returncode from child (AHanses).
        */
        void waitforChild();
#endif
};

```

implementation module

```

#include <config.h>

#ifdef __GNUG__
#pragma implementation
#endif

#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <process.h>
#include "syscall.h"
#include "syscontr.h"

//-----
// Class, which controls a system-call
//-----

// constructor
Systemcalls::Systemcalls()
{
    pid = (pid_t) 0; // yet no child
}

// constructor
//
// starts child
Systemcalls::Systemcalls(Starttype how, LString what, Callbackfct cback)

```

```

{
    start    = how;
    command = what;
    cbk      = cback;
    pid      = (pid_t) 0; // no child yet
    retval   = 0;
    Startscript();
}

// destructor
// not yet implemented (?)
Systemcalls::~Systemcalls()
{
}

// Start a childprocess
//
// if child runs in background, add information to global controller.

int Systemcalls::Startscript()
{
    retval = 0;
    switch (start) {
    case System:
        retval = system(command.c_str());
        Callback();
        break;
    case Wait:
        pid = Fork(P_WAIT);
        if (pid>0) { // Fork succesful. Wait for child
#ifdef __EMX__
            waitForChild();
#endif
            Callback();
        } else
            retval = 1;
        break;
    case DontWait:
        pid=Fork(P_NOWAIT);
        if (pid>0) {
            // Now integrate into Controller
            SystemcallsSingletoncontroller::Startcontroller starter;
            SystemcallsSingletoncontroller *contr=
                starter.GetController();
            // Add this to contr
            contr->AddCall(*this);
        } else
            retval = 1;
        break;
    //default: // error();

```

```

        //break;
    }
    return retval;
}

// Wait for child process to finish. Returns returncode from child.
#ifdef __EMX__
void Systemcalls::waitForChild()
{
    // We'll pretend that the child returns -1 on all errorconditions.
    retval = 1;
    int status;
    bool wait = true;
    while (wait) {
        pid_t waitrpid = waitpid(pid, &status, WUNTRACED);
        if (waitrpid == -1) {
            perror("LyX: Error waiting for child");
            wait = false;
        } else if (WIFEXITED(status)) {
            // Child exited normally. Update return value.
            retval = WEXITSTATUS(status);
            wait = false;
        } else if (WIFSIGNALED(status)) {
            fprintf(stderr, "LyX: Child didn't catch signal %d "
                "and died. Too bad.n", WTERMSIG(status));
            wait = false;
        } else if (WIFSTOPPED(status)) {
            fprintf(stderr, "LyX: Child (pid: %ld) stopped on "
                "signal %d. Waiting for child to finish.n",
                (long) pid, WSTOPSIG(status));
        } else {
            fprintf(stderr, "LyX: Something rotten happened while "
                "waiting for child %ldn", (long) pid);
            wait = false;
        }
    }
}
#endif

// generate child in background

pid_t Systemcalls::Fork(int spawnFlag)
{
#ifdef __EMX__
    pid_t cpid=fork();
    if (cpid == 0) { // child
#endif
        LString childcommand(command); // copy
        LString rest = command.split(childcommand, ' ');

```

```

        const int MAX_ARGV = 255;
        char *syscmd = NULL;
        char *argv[MAX_ARGV];
        int index = 0;
        bool Abbruch;
        do {
            if (syscmd == NULL) {
                syscmd = childcommand.copy();
            }
            argv[index++] = childcommand.copy();
            // reinit
            Abbruch = !rest.empty();
            if (Abbruch)
                rest = rest.split(childcommand, ' ');
        } while (Abbruch);
        argv[index] = NULL;
        // replace by command. Expand using PATH-environment-var.
#ifdef __EMX__
        execvp(syscmd, argv);
        // If something goes wrong, we end up here:
        perror("LyX: execvp failed");
    } else {
#else // spawn child in another process; fork() is not portable
        pid_t cpid = spawnvp(spawnFlag, syscmd, argv);
#endif
        if (cpid < 0) // error
            perror("LyX: Could not fork");
#ifdef __EMX__
    } else // parent
#endif
        return cpid;
    return 0;
}

// Reuse of instance

int Systemcalls::Startscript(Starttype how, LString what, Callbackfct cback)
{
    start    = how;
    command  = what;
    cbk      = cback;
    pid      = (pid_t) 0; // yet no child
    retval   = 0;
    return Startscript();
}

//

```

```

// Mini-Test-environment for script-classes
//
#ifdef TEST_MAIN
#include <stdio.h>

int SimulateTimer;
void back(LString cmd, int retval)
{
    printf("Done: %s gave %dn", cmd.c_str(), retval);
    SimulateTimer = 0;
}

int main(int, char**)
{
    SystemcallsSingletoncontroller::Startcontroller starter;
    SystemcallsSingletoncontroller *contr=starter.GetController();

    Systemcalls one(Systemcalls::System, "ls -ltag", back);
    Systemcalls two(Systemcalls::Wait, "ls -ltag", back);
    SimulateTimer = 1;
    Systemcalls three(Systemcalls::DontWait , "ls -ltag", back);
    // Simulation of timer
    while (SimulateTimer)
        {
            sleep(1);
            contr->Timer();
        }
}
#endif

```

10.4 Multithreading instead of fork()/vfork()

[to be written]

10.5 Unix file system issues

The separator of path components is a slash '/' as opposed to OS/2's '. Note that many APIs understand the '/' on OS/2, but cmd.exe and most apps don't. Multiple paths (e.g. in environment variables like PATH) are separated by a colon ':'. The root dir on every un*x system is '/'. Often code checks whether a given path is an absolute one by comparing the beginning with '/'. Use grep for all standard paths and all commands which are mentioned in this introduction. (`_fullpath()` and `_abspath()` can also be useful.)

Using

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

may help to support drive letters(`_fullpath()` and `_abspath()` can also be useful). Read the emx docs:

"Watch out for Unix file system hacks: Unix allows deleting and renaming an open file (the

- Watch out for Unix file names (Unix is case sensitive, long file names and multiple dots are allowed). On OS/2's HPFS multiple dots are also allowed; however, trailing dots are not significant (except for the special file names '.' and '..').

- 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"`, respectively. However,

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

won't work as the standard OS/2 and DOS command interpreters don't recognize `/dev/null`.

- If you want to use for separating directories, changes may be necessary. These changes are optional because `/` also works. "

"Unix does not have drive letters, so a file (resources, fonts, config, etc.) path starts with '/' normally. XFree86 is located in a tree below `%X11ROOT%\XFree86...`. To prepend the `X11ROOT` part to a file path, a special function named `__XOS2RedirRoot` exists which you are supposed to use in these cases rather than an insane own solution. The prototype of this function is:

```
const char *__XOS2RedirRoot(const char *pathname);'
```

[...] XFree86/OS2 stores all configuration files in a common directory. This is `X11ROOT:\XFree86\lib\X11`, where `X11ROOT` is an environment variable which contains the drive letter the directory tree is located. The common places like `/usr/X11`, `/usr/X11R6`, `/usr/XFree86`, `/etc` which are used for XFree86 in various Unix flavors, are not supported [...]. It is also there where you should put your `XF86Config` file, the color database (`rgb.txt`), the host authorization files (`X0.hosts`), and various other files." (FAQ)

"Potential problem might be the DLL name: IIRC one can either use

```
'foo' or 'x:\barfoo.dll'
```

to access `foo.dll` (if in `LIBPATH`).” (Alexander Mai)

”Note that `///abc` is a valid Unix filename. It’s equivalent to `/abc`.

- Note that `chdir (“..”)` is a no-op under Unix if the current working directory is the root directory. Under `emx`, `chdir (“..”)` fails in the root directory.

- Use `termio` or `termios` or read the keyboard with `_read_kbd()` if you don’t want to get input line by line.

- Under Unix, directories in environment variables (`PATH`, for instance) are separated by colons; use semicolons instead.

- The shell isn’t called `/bin/sh`. Use `system()`. `system()` and `popen()` don’t expand wildcards (unless `COMSPEC` points to a shell which expands wildcards).”

So, in short:

”grep for `’/dev’`, `’/bin’`, `’/tmp’`, `’/usr’`, `’/proc’`, ... also check out all commands starting up another process:

`exec*`, `popen`, `system`,...

Remember that `cmd.exe` doesn’t like slashes!” (Alexander Mai)

10.6 Sockets

”`emx` includes support for sockets of IBM TCP/IP for OS/2 (including the Internet Access Kit of OS/2 3.0). The headers files are derived from NetBSD and are packaged in `bsddev.zip`. Some manual pages are packaged in `bsddev.zip`.”

Use the `-lsocket` option to link with the socket library. Note that `-lsocket`, like all `-l` option, should be given after the source files:

```
gcc mymain.c mysub.c -lmylib -lsocket -lm
```

“`emx` tries to provide seamless support for sockets, like Unix systems do. This approach is quite different from the one taken by IBM’s TCP/IP for OS/2 Toolkit.

Sockets are not supported with the system call library `sys.lib` (`-Zsys`).

The header `<sys/so_ioctl.h>`, which includes TCP/IP-specific definitions for `ioctl()`, is included by `<sys/ioctl.h>` only if a TCP/IP-related header such as `<sys/socket.h>` has been included before `<sys/ioctl.h>`. If you don’t want to reorder the `#include` directives, define the symbol `_EMX_TCPIP` before including `<sys/ioctl.h>`, or include `<sys/so_ioctl.h>` directly (however, the last alternative is not recommended).

When porting Unix applications, please note the following restrictions:

Socket handles are not inherited across `exec*()` and `spawn*()`. (Note that a process created by `fork()` inherits the socket handles of its parent process.) Inheriting socket handles is planned for a future release of `emx`.

The size of messages is restricted to 32767 bytes (this is a limitation of IBM TCP/IP for OS/2).

Initially, 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."

Look at `emx' netdb.h`, where you will find an excerpt of the `bsd paths.h` mechanism, but hidden behind your back and without any further explanation what is going on:

```
#define _PATH_HEQUIV "/tcpip/etc/hosts.equiv"
#define _PATH_HOSTS "/tcpip/etc/hosts"
#define _PATH_NETWORKS "/tcpip/etc/networks"
#define _PATH_PROTOCOLS "/tcpip/etc/protocols"
#define _PATH_SERVICES "/tcpip/etc/services"
```

"The difficulty here is that one still needs to annotate each of these paths with a drive letter. `XFree86` has to do this with almost any internal path by using a routine named `__XOS2RedirRoot` which prepends the `%X11ROOT%` variable. Here is the drawback of DOS like systems to behave like VMS ("everything is a device, and some of them, denoted by drive letters can contain file systems") in contrast to Unix philosophy ("everything is a file, and some of them may have special properties, like being a pipe, or a socket, or a device")." (Holger Veit <Holger.Veit@gmd.de>)

10.7 Environment (getenv())

EDITOR Your favourite editor. In case you don't want to end up inside `vi` [a famous `un*x` editor] and don't know how to get out `:q!`

HOME points to a directory which contains user specific data. Always non-NULL on `un*x`

PAGER standard tool to page output. Something like `"more"` or `"less"`. Use `xless` in `XFree`.

PATH same as on OS/2. Note to use `';`' instead of `':'`.

PRINTER default printer. Might be used in conjunction with printing tools like `"lp"`, `"lpr"`.

SHELL standard shell to be used. Often used by Makefiles, ...

TERM Name of the terminal currently used. (see `termcap`, `terminfo`). Make sure your software runs in an OS/2 window as well as in an xterm.

TMP Usually not necessary, since all systems have a directory for temporary stuff in `/tmp`. If you don't have `'tmp'`, create it! Many software depends on this.

WINDOWID Id of current X11 window.

10.8 Miscellanea

Unfortunately many OS/2 tools, including built-in `cmd.exe` commands have their counterparts on `un*x` with exactly the same name but different semantics/syntax. An incomplete list: `echo`, `find`, `patch`, `sort`.

" By default, signal processing is different when using `signal()`: `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` options of GCC. If you use POSIX.1 functions for signal handling, `SIG_ACK` is not required.

- 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`."

Part III

Appendices

Chapter 11

Some Notes on Portability and Coding Style

SMiyata in a contribution to the XFree86/OS2 mailing list pointed out:

Having as many softwares as possible really helps even if their qualities are sometimes questionable. On the other hand, for example, I am using L_yX for a practical purpose so that if the reliability of it is lower than that on some other platform, say, LINUX, I will eventually be forced to switch the operating system I use daily. And (more or less) functional ports do not appear over one night. I myself cannot grasp the contents of the source code at a glance. I have to test semi-working versions and investigate the relevant parts of the source again and again to have something working. [...] I was really helped that most of the non-working features wererather evident. I could pin-point the relevent parts of the source easily.

The problem of automating dirty workaround stuffs is that unless the one who ports a software awares that s/he is doing something dirty, it is really likely for him/her to let the semi-working workarounds unnoticed.

There are at least two remedies for this problem.You can let the dummy stub functions to print warning messages [...], although this may alarm also the users of the port. Another possiblity is to make the warning very clear in the doc.

Alexander Mai wrote:

Or do it like other projects: Use an environment variable to switch on/off those warnings. Might even include multiple levels, one printing statements once and another which reminds everytime of some missing functionality.

Write Proper Implementations.**Example: `cpfile.c`, a widely portable replacement for `link()/symlink()`.**

Alexander Mai wrote:

I'm not sure at all what people think here, but I make some proposals ... :

- a) use a `file_copy` routine for `symlink` (and don't clean up later perhaps)
- b) let `getdtablesize` return some 'guessed' integer (w/o checking any docs, just get it working in most cases ...)
- c) `#define lstat stat` (neglecting something one could imagine around TVFS)

Here is a `file_copy`- like routine, that has been discussed on the `emx` mailing list. Diagnostics in case of errors should be written to a `error.log` file. In an application program this can be achieved by redirecting the output of `stdout` to a file (you may use `freopen()` in your application's data directory:

```
/**** cpfile.c: cpfile() ***
```

```
From: "mithra" <mithra@bga.com>
To: emx@queue.iaehv.nl
Subject: Re: COPY/MOVE system function?
```

Since I needed a copy I could use everywhere without invoking `system()`, here is `cpfile.c`:

```
****
* Copy file from 'in_name' to 'out_name'
*
* return:
* 0 if all's well
* 1 if can't open in_name
* 2 if can't open out_name for writing
* 3 if if stat() in_name returns an error
* 4 if copy failed (read error)
* 5 if copy failed (write error)
* 6 if out_name close() failed (file system error!)
* Check to prevent silent data loss, e.g. with network data.
* 7 if utime() failed (time set error)
* 8 if chmod() of output file failed
* 9 if in_name close() failed (file system error!)
*
* in future may want access status of output file...
* Curtis W. Rendon 12/20/1999 initial
```

```

* modified 04/01/00 A. Hanses
*
* hereby released to the public domain, please keep my comments,
* no warranty inferred, implied, or specified.
* Curtis W. Rendon mithra@earthling.net 12/29/1999
*
**** Prototype: */

int cofile (const char *in_name, const char *out_name);

#include <stdio.h>

#ifdef VIOLATE_POSIX_USE_M$_EXTENSION
# include <io.h> /* PC extension, not (Posix) portable */
#else
# include <unistd.h> /* Emx has portable (Posix) headers */
#endif

#include <utime.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h> /* ANSI/ISO C forbids explicit declaration ! *
  It is a thread local modifiable (int) lvalue (usually a macro) */

#ifndef lstat /* This is a file system dependency */
#define lstat(x,y) stat(x,y)
#endif

static FILE *out_p; /* This is the pointer to the new or trun- *
  cated old output file. Input file attributes are also copied. */

static struct stat file_status_buf; /* File attributes buffers */
static struct utimbuf file_utime_buf;

static char buf[BUFSIZ]; /* r/w no more than BUFSIZ at once */
static size_t count; /* read/write at least count bytes at once */

static int status = 0; /* default return code */

int
cofile(const char *in_name, const char *out_name)
{
  FILE *const in_p = fopen(in_name, "rb"); /* point to in file */

  if (status) /* reset and try your luck again, if a previous */
    status = 0; /* call to cofile() has returned an error */

  if (in_p == NULL) /* error opening in file */
    return (status = 1);
}

```

```

if ((access(out_name, W_OK) == 0) /* overwriting out file */
    fprintf(stderr, "Warning: Overwriting '%s'...\n", out_name);
else if ((access(out_name, F_OK) == 0) {
    if (fclose (in_p) == EOF)
        perror("cpfile.c: Error closing input file");
    fprintf(stderr, "Error: Cannot write to '%s'\n", out_name);
    return (status = 2); /* error out file not writable */
} /* Note: Posix' link()/symlink() don't overwrite! To
 * achieve an equivalent behaviour, you can test for
 * W_OK and then return()
 */
if ((out_p = fopen(out_name, "wb")) == NULL)
    status = 2; /* error opening out file */
else {
    if (lstat(in_name, &file_status_buf) == -1) {
        status = 3; perror("cpfile.c: Input file stat() error");
    }
    do { /* endless file i/o loop */
        count = fread( (void*) buf, 1, sizeof(buf), in_p);

        if ( (count < sizeof(buf)) && ferror(in_p) ) {
            status = 4; perror("cpfile.c: Internal read error");
            break;
        }
        if (fwrite( (void*) buf, 1, count, out_p) < count) {
            status = 5;
            perror("cpfile.c: Internal write error");
            break;
        }
    } while ( ! feof(in_p) ); /* end file i/o loop */

    if (fclose (out_p) == EOF)
        return (status = 6); /* Now in danger of data loss ! */

    file_utime_buf.actime = file_status_buf.st_atime;
    file_utime_buf.modtime = file_status_buf.st_mtime;

    if (utime(out_name, &file_utime_buf) == -1) {
        status = 7; perror("cpfile.c: utime() error");
    }
    if (chmod(out_name, file_status_buf.st_mode) == -1) {
        status = 8; perror("cpfile.c: chmod() error");
    }
}
if (fclose (in_p) == EOF)
    status = 9;

return (status);
}

```

```

#ifdef DEBUG
int main(const int argc, const char *const argv[])
{
    if (argc != 3) {
        puts("Usage: cpfile INFILE OUTFILE");
        exit(-1);
    }
    errno = 0;
    if (cpfile(argv[1], argv[2])) {
        fprintf(stderr, "%s %s %s: Errorlevel: %d\n",
            argv[0], argv[1], argv[2], status);
        if (errno) perror("cpfile.c");
    }
    return (status);
}
#endif

```

On Tue, 4 May 1999 10:44:58 +0200, Holger Veit wrote:

Macros can either be used to hide obscure hacks, or making the code more easily to understand. This depends on the situation. `#ifdef` clauses, namely nested ones are ill, ill, ill. Consider the following fragment in `Xos.h`:

```

#ifdef USG
#ifdef __TYPES__
#ifdef CRAY
#define word word_t
#endif /* CRAY */
#include <sys/types.h> /* forgot to protect it... */
#define __TYPES__
#endif /* __TYPES__ */
#else /* USG */
#if defined(_POSIX_SOURCE) && (defined(MOTOROLA) || defined(AMOEBA))
#undef _POSIX_SOURCE
#include <sys/types.h>
#define _POSIX_SOURCE
#else
#include <sys/types.h>
#endif
#endif /* USG */

```

The X11 headers have many more of these 'pearls', and the GNU glibc include headers are even worse. The only way to find out what really goes into the code is to run `gcc -E` on the file that includes this file.

```

> > b) a quick hack to satisfy compiler/linker
>
> using a function stub foo for a missing function:
>
> foo()
> {
>     fprintf (stderr, "sorry, foo() not implemented! I'm just too lazy!
> You may consider this as a bug; but I won't care at all ;-)" );
>     return -1;
> };
>
> > and

```

```

> > c) a proper implementation
>
> Providing all this ifdef'd in the sources (no stubs) and doing it
> according to the books in a way that will always work and is easy to
> debug.

```

You can't prevent laziness in user code; I myself have routines marked with

```
#define TBD(x) fprintf(stderr, #x ": not yet implemented\n")
```

The point is rather: missing functions, namely prominent things like 'link', 'readlink', 'chown', 'setuid' and others should be moved into a separate OS-dependent file, rather than handled in the following way:

```
(config.h)
#define _HAS_CHOWN 0
```

```
(foo.c)
#if _HAS_CHOWN
    chown(blurb...);

```

```
#else
    /*do nothing*/
#endif
```

scattered all over the place. It is just the problem that people don't learn at all (even not in CS classes at universities) how to practically write portable code. Code is still written in a plumbing manner: Add some pipes in some arbitrary house so that the water just flows through it, and if you need another kitchen sink somewhere, just add a flange in a place wherever your editor goto-line command accidentally points you to. Not even talking about Booch's OOA/OOD/OOP or other techniques of software analysis and design, but the drawback of bazaar programming is distributed hacking and patching.

>

> Well, here I'm too inexperienced.

II. Provide a LIBERTY.DLL

We'd consider making some libiberty.a (adding at least stubs for all routines that are missing in EMX or are implemented differently in Unix dialects) a definite standard. The XFree86-4.0 server now for instance also needs ndbm and regex, for some reasons. I had to take this from elsewhere." (Holger Veit)

"Not #ifdef but implement! Or at least provide stubs, to go without the need to #ifdef. #ifdef's when not maintained tend to rust much faster than code without them. And with such core-level thing as XFree86 the time to live is essential." (Sergey Ayukov)

III. Provide a BSD-like Paths.h Mechanism

```

> #define _PATH_HEQUIV  "/tcpip/etc/hosts.equiv"
> #define _PATH_HOSTS   "/tcpip/etc/hosts"
> #define _PATH_NETWORKS "/tcpip/etc/networks"
> #define _PATH_PROTOCOLS "/tcpip/etc/protocols"
> #define _PATH_SERVICES "/tcpip/etc/services"
>
> Well, how do we get things mapped to the current values here?
>
> E.g. there are lots of sources assuming an existing '/tmp'.
> Should we have an EMX internal variable/function as
> a replacement?

```

That's what a "Common Filesystem Standard" was supposed to be good for. EMX or any other redirection software (e.g. TVFS) is then responsible to map such paths to really existing locations. The above paths.h snippet implies that there is some directory, from now on known as "/tcpip/etc" which contains two files "protocols" and "services". TVFS might redirect *this directory* to e.g. "/mptn/etc" but should not tear the files apart resulting in obscure to find bugs (probably not in this case, but situations are possible where the definition of such well-known paths is important)."

On Tue, 04 May 1999 21:36:19 -0100, Martin Krischik wrote:

how about:

```

typedef enum pathId
{
    pathHEQUIV    ,
    pathHOSTS    ,
    pathNETWORKS ,
    pathPROTOCOLS ,

```

```

    pathSERVICES    ,
    } pathId;

extern void
pathCalc(unsigned char aPath[],
          enum PATH aID);

```

Would that not be a lot cleaner version. The Function would calculate the correct filepath from the correct environment variable and filename.

IV. Beware of Global Variables

On Thu, 21 Oct 99 14:44:43, Asbjoern Pettersen wrote:

```

>When i started GIMP porting a long time ago 0.9.18 i run into this problems.
>The main problem is use of global variables and forking() of other processes
(DLL
>loading).
>They use global variables as input !! That have never been a good design on
any
>OS'es.
>So the problem is how to set (and find) those global variables !
>This input is hidden and a new programmer will not know the real impact on
changes
>he do. It's terrible! The program will get buggy and out of control.

```

V. Try Hard to Respect Posix (and ANSI/ISO C/C++)

Remember: Emx is very close to ANSI/ISO and Posix.

- * Avoid common GNU extensions.
- * Don't cling to vendor-specific so-called 'standards'.
- * Follow closely internationally accepted independent portability standards.

This means that emx programmers are sometimes reminded of the fact that 'Posix does not buy you much...'. This is often a headache, but it also shows you which code is portable, aka standard conformant, and which code isn't, therefore has to add some quirks to gain portability:

On 06 Sep 1999 20:22:17 +0200, Lars Gullik Bjnnes wrote:

>Duncan Simpson <dps@io.stargate.co.uk> writes:

>| According to my man page strcasecmp is conforming to BSD 4.3 (and
>| all unicies I have seen so far, inclduign SunOS). AFAIK GNU libc 2.1 is
>| conforming to ANSI and Posix but also includes common extensions, like
mmap,
>| lstat, setrlimit, snprintf, select, fileno, *BSD sockets and whole bunch of
>| other stuff that is endemic in un*x source code and supported everywhere
with
>| very few exceptions.
>| I can use fdopen in Posix programs. There are also some things that only
exist
>| in glibc, and linux libc 5, like strfry(3) and memfrob(3), which are presumably
>| included for hack value.
>|
>| The average punter, and Un*x programmer, is not aware of how little Posix
>| gives him. As soon as you prevent a remote root exploit by using snprintf
you
>| have violated Posix
>

>And you don't know if your prog will compile on a system not running gnu libs.

>

>|—and of course by doing any networking and using

>| select(2) you have violated Posix anyway. All internationalisation is beyond

>| the Posix standard too.

>

>Not quite true, check man setlocale:

>CONFORMING TO

> ANSI C, POSIX.1

>We do not want to use strcasecmp, and in L^yX 1.1.x we are getting rid

>of it. (using more native C++ constructs instead.)

"As for non-existing API, I would recommend to stick to well-known things

like Posix, BSD networking, X11 and avoid system-specific calls like DosWrite() etc. completely. If you're doing really multiplatform project. Really 'standard' calls are the same everywhere and rarely require re-implementation. There's some oddity here and there (I heard Ultrix(?) does not have strdup(?) but you can add it in transparent way, namely linking on Ultrix with your own implementation of strdup()."

(Dr. Sergey Ayukov, Sternberg Astronomical Institute Moscow, Russia

<http://www.ayukov.com>, <http://crydee.sai.msu.ru/index-asv.html>)

"Ok, can someone explain (together with an example each) what is

a) a dirty trick, b) a quick hack to satisfy compiler/linker c) a proper implementation"

On Tue, 4 May 1999 10:44:58 +0200, Holger Veit wrote:

> >Ok, can someone explain (together with an example each) what is > > >
> a) a dirty trick, > > using a preprocessor macro instead of #ifdef in the sources, so that > you can watch in place what is going on (if things go wrong; remember: > Everything that can go wrong will go wrong!)

Macros can either be used to hide obscure hacks, or making the code more easily to understand. This depends on the situation. #ifdef clauses, namely nested ones are ill, ill, ill. Consider the following fragment in Xos.h:

```
#ifndef USG #ifndef __TYPES__ #ifdef CRAY #define word word_t #endif
/* CRAY */ #include <sys/types.h> /* forgot to protect it... */ #define
__TYPES__ #endif /* __TYPES__ */ #else /* USG */ #if defined(_POSIX_SOURCE) && (defined(MOTOROLA) || defined(AMOEBA))
#undef _POSIX_SOURCE #include <sys/types.h> #define _POSIX_SOURCE
#else #include <sys/types.h> #endif #endif /* USG */
```

The X11 headers have many more of these 'pearls', and the GNU glibc include headers are even worse.

The only way to find out what really goes into the code is to run `gcc -E` on the file that includes this file.

```
>> b) a quick hack to satisfy compiler/linker >> using a function stub foo for
a missing function: >> foo() { > fputs(stderr, "sorry, foo() not implemented!
I'm just too lazy! > You may consider this as a bug; but I won't care at all ;-)"
); > return -1; > }; >>> and
```

```
>> c) a proper implementation >> Providing all this ifdef'd in the sources
(no stubs) and doing it > according to the books in a way that will always work
and is easy to > debug.
```

You can't prevent laziness in user code; I myself have routines marked with

```
#define TBD(x) fprintf(stderr, #x ": not yet implementedn")
```

The point is rather: missing functions, namely prominent things like 'link', 'readlink', 'chown', 'setuid' and others should be moved into a separate OS-dependent file, rather than handled in the following way:

```
(config.h) #define _HAS_CHOWN 0
```

```
(foo.c) #if _HAS_CHOWN chown(blurb...); #else /*do nothing*/ #endif
```

scattered all over the place.

```
> But for great projects this is simply not feasible even if investing a > reason-
able amount of time.
```

It is. It is just the problem that people don't learn at all (even not in CS classes at universities) how to practically write portable code. Code is still written in a plumbing manner: add some pipes in some arbitrary house so that the water just flows through it, and if you need another kitchen sink somewhere, just add a flange in a place wherever your editor goto-line command accidentally points you to. Not even talking about Booch's OOA/OOD/OOP or other techniques of software analysis and design, but the drawback of bazaar programming is distributed hacking and patching.

```
>> I'm not sure at all what people think here, but I make some proposals ... :
>>> a) use a file_copy routine for symlink (and don't clean up later perhaps)
>> yes this is missing >>> b) let getdtablesize return some 'guessed' integer
(w/o checking any >> docs, just get it working in most cases ...) >> c) #define
lstat stat (neglecting something one could imagine around TVFS) >> Well,
here I'm too inexperienced.
```

II. Provide a LIBERTY.DLL _____

We'd consider making some libiberty.a (adding at least stubs for all routines that are missing in emx or are implemented differently in Unix dialects) a definite standard. The XFree86-4.0 server now for instance also needs ndbm and regex, for some reasons. I had to take this from elsewhere." (Holger Veit)

"Not #ifdef but implement! Or at least provide stubs, to go without the need to #ifdef. #ifdef's when not maintained tend to rust much faster than code without them. And with such core-level thing as XFree86 the time to live is essential." (Sergey Ayukov)

III. Provide a BSD-like Paths.h Mechanism —————

```
> #define _PATH_HEQUIV "/tcpip/etc/hosts.equiv" > #define _PATH_HOSTS
"/tcpip/etc/hosts" > #define _PATH_NETWORKS "/tcpip/etc/networks" >
#define _PATH_PROTOCOLS "/tcpip/etc/protocols" > #define _PATH_SERVICES
"/tcpip/etc/services" > > Well, how do we get things mapped to the current
values here? > > E.g. there are lots of sources assuming an existing '/tmp'. >
Should we have an emx internal variable/function as > a replacement?
```

That's what a "Common Filesystem Standard" was supposed to be good for. emx or any other redirection software (e.g. TVFS) is then responsible to map such paths to really existing locations. The above paths.h snippet implies that there is some directory, from now on known as "/tcpip/etc" which contains two files "protocols" and "services". TVFS might redirect *this directory* to e.g. "/mptn/etc" but should not tear the files apart resulting in obscure to find bugs (probably not in this case, but situations are possible where the definition of such well-known paths is important)."

On Tue, 04 May 1999 21:36:19 -0100, Martin Krischik wrote:

how about:

```
typedef enum pathId { pathHEQUIV , pathHOSTS , pathNETWORKS , path-
PROTOCOLS , pathSERVICES , } pathId;
```

```
extern void pathCalc(unsigned char aPath[], enum PATH aID);
```

Would that not be a lot cleaner version. The Function would calculate the correct filepath from the correct environment variable and filename.

IV. Beware of Global Variables —————

On Thu, 21 Oct 99 14:44:43, Asbjørn Pettersen wrote:

```
>When i started GIMP porting a long time ago 0.9.18 i run into this problems.
>The main problem is use of global variables and forking() of other processes
(DLL >loading). >They use global variables as input !! That have never been
a good design on any >OS'es. >So the problem is how to set (and find) those
global variables ! >This input is hidden and a new programmer will not know
the real impact on changes >he do. It's terrible! The program will get buggy
and out of control.
```

V. Try Hard to Respect Posix (and ANSI/ISO C/C++) —————

Remember: Emx is very close to ANSI/ISO and Posix.

* Avoid common GNU extensions.

* Don't cling to vendor-specific so-called 'standards'.

* Follow closely internationally accepted independent portability standards.

This means that emx programmers are sometimes reminded of the fact that 'Posix does not buy you much...'. This is often a headache, but it also shows you which code is portable, aka standard conformant, and which code isn't, therefore has to add some quirks to gain portability:

On 06 Sep 1999 20:22:17 +0200, Lars Gullik Bjnnes wrote:

>Duncan Simpson <dps@io.stargate.co.uk> writes:

>| According to my man page strcasecmp is conforming to BSD 4.3 (and >| all unicies I have seen so far, inclduign SunOS). AFAIK GNU libc 2.1 is >| conforming to ANSI and Posix but also includes common extensions, like mmap, >| lstat, setrlimit, snprintf, select, fileno, *BSD sockets and whole bunch of >| other stuff that is endemic in un*x source code and supported everywhere with >| very few exceptions. >| I can use fdopen in Posix programs. There are also some things that only exist >| in glibc, and linux libc 5, like strfry(3) and memfrob(3), which are presumably >| included for hack value. >| >| The average punter, and Un*x programmer, is not aware of how little Posix >| gives him. As soon as you prevent a remote root exploit by using snprintf you >| have violated Posix > >And you don't know if your prog will compile on a system not running gnu libs. > >|—and of course by doing any networking and using >| select(2) you have violated Posix anyway. All internationalisation is beyond >| the Posix standard too. > >Not quite true, check man setlocale: >CONFORMING TO > ANSI C, POSIX.1

>We do not want to use strcasecmp, and in LyX 1.1.x we are getting rid >of it. (using more native C++ constructs instead.)

"As for non-existing API, I would recommend to stick to well-known things like Posix, BSD networking, X11 and avoid system-specific calls like DosWrite() etc. completely. If you're doing really multiplatform project. Really 'standard' calls are the same everywhere and rarely require re-implementation. There's some oddity here and there (I heard Ultrix(?) does not have strdup(?) but you can add it in transparent way, namely linking on Ultrix with your own implementation of strdup()."

(Dr. Sergey Ayukov, Sternberg Astronomical Institute Moscow, Russia <http://www.ayukov.com>, <http://crydee.sai.msu.ru/index-asv.html>)

Preface

A. Instead of an Introduction

I. Where to Start?

II. Some Frequently Used yet Uncommon Concepts

1. Another Interesting Name: What's 'X'?
2. And Posix, etc.?
3. And to How Get Posix, etc. Defined?

B. A Silly Demo and Some Comments

- I. Source with Comments
- II. Module Definition File

C. Notes on Libraries

- I. Dynamic link libraries (DLL)
- III. Shared libraries and dynamic libraries (TOG UNIX V.1/2)
- IV. Exporting symbols from a DLL (OS/2 OMF)
- V. 'dynamic library' and 'shared object' (*.so) versus DLL
 - 1. dynamic link library
 - 2. 'dynamic library' and 'shared object' (*.so) versus DLL
 - 3. demo
- VI. How to Link
- VII. Some Special Tricks for libraries

D. Now the Hardware Gets Involved

- I. Devices, Servers, Clients
 - 1) '/dev/fastio\$', '/dev/console\$', '/dev/ptyP0\$', etc.
 - 2) concepts related to hardware (vio) access and kernel level API 's
- II. Inter Process (Session) Communication
- III. Emx and device driver programming (hardware devices)

E. 'Imakefile' or 'configure', that's here the question!

- I. A short overview: Autoconf vs. Imakefile
- II. Preliminary steps
 - 1) RTF docs
 - 2) Some additions to /XFree86/lib/X11/config/os2.cf and os2.rules
- III. Common initial problems
 - 1) 'xmkmf -a' crashes because of 'sh' and 'Rexx' (path name) conflicts
 - 2) 'make' crashes because of 'sh' and 'Rexx' conflicts
- IV. Imakefile examples
 - 1) creating a static library (archive)
 - 2) creating an executable
 - 3) Some maybe useful defines

E. What Shall I Better Change in the Sources?

I. Globbing

II. DOS "text" mode problems and binary reading

III. Process management and fork()

1) What is the difference between fork() and spawnvp()?

a) fork() and emx

b) making it work

c) replacing fork() and execvp(), etc.

2) Example

IV. Unix file system issues

V. Sockets

VI. Miscellanea

F. Some Notes on Portability and Coding Style

I. Write Proper Implementations

II. Provide a LIBERTY.DLL

III. Provide a BSD-like Paths.h Mechanism

IV. Beware of Global Variables

V. Try Hard to Respect Posix (and ANSI/ISO C/C++)

3) How to Link

I recommend the following simple (unorthodox, but nearly 'foolproof') procedure:

Use the static archive provided on the net or use the Makefile (normally created by Imake, but today more and more by the OS/2 port of autoconf; details later) to make the static archive foo_s.a. Then:

1. 'emxomf -s -l foo_s.a' (convert to native omf format)

2. 'emximp -o foo.a foo.def' (create import library for later ld linker fixup)

3. 'emxomf -s -l foo.a' (convert to native omf format for link386)
4. 'gcc foo.def -o foo.dll -Zomf -Zdll -Zmtd -Zsysv-signals -Zbin-files
 -Zlinker /EXEPACK:2 -Zlinker /NOI -Zlinker /NOL -Zlinker /NOD -s -fstack-check
 -Zlinker /PACKDATA -lfoo_s -lmore_foo_libs_here -lXmu -lXt -lSM -lICE
 -lX11
 -lXext -lbsd -lsocket'

This produces a packed dll. While some linker flags aren't really necessary, they won't hurt, unless you use OS/2 2.x (use '/EXEPACK' if in doubt). Isn't the commandline nice?

Note #1 (linking order):

Linking order is crucial! The traditional C-style linker must find the fixup for still unresolved symbols in the same library or in another one which is `_following_` on the command line. You might even need to link in rare cases like:

```
-lfoo1 -lfoo2 -lfoo1
```

where foo1 needs symbols exported by foo2 and foo2 needs symbols exported by foo1.

Note #2 (library debugging):

For a debuggable library compiled with '-g' you'd use the line, except '-Zomf -s' and all the '-Zlinker blurb', which are specific to link386.

Note #3 (XFree86/2 and multi-threading):

Note #4 (recommended flags for speed):

(emx0.9c)

```
CXXFLAGS = -m486 -O3 -Zmtd -Zsysv-signals -D__ST_MT_ERRNO__  
-Wall -malign-loops=2  
-malign-jumps=2 -malign-functions=2 -ffast-math
```

```
LDFLAGS = -s -Zmtd -Zsysv-signals -Zbin-files
```

[your mileage may vary; add your testing results here!]

Trick#3: (On Sun, 17 Oct 1999 14:34:21 +0200 (MET DST), Stefan Neis wrote):

Hint: Use instead ANSI C ('stdio.h') (or Posix ('unistd.h')) interfaces!

"3.11

Can I use another shell like 'bash' in place of CMD.EXE in an xterm?
Most shells are subtly incompatible with the PTY mechanism used in
xterm. A symptom may be that they won't work either if you redirect
console I/O to a COM terminal which should be possible. bash is
such an incompatible shell. See also Q 3.12 and Q 3.13.

Meanwhile, there are special ports of tcsh and ksh which work in
an xterm.

3.12

I use 4OS/2 (or some other shell) as my shell. Will it work in xterms?
4OS2 has a number of misfeatures, politely spoken, in an xterm. We
attempt to fix this in the future. For now, don't use it in an an xterm.
If 4OS2 or another shell is in your OS2_SHELL or SHELL variable,
please add an environment variable named X11SHELL to point to a
valid CMD.EXE path, e.g.

```
SET X11SHELL=D:OS2CMD.EXE
```

in your CONFIG.SYS. This will override the OS2_SHELL or SHELL setting which is used otherwise.

3.13

Why does OS2MORE.COM give a SYS0447 in an xterm?

This is a program which silently assumes it has access to the keyboard and the screen: it uses functions from KBDCALLS, MOUCALLS, or VIOCALLS. There are more programs of this kind, for instance: most *.COM in os2, ATTRIB.EXE, BACKUP.EXE, CACHE.EXE, EAUTIL.EXE, FIND.EXE, HELPMSG.EXE, LINK.EXE, LINK386.EXE(!), PATCH.EXE, PSTAT.EXE, REPLACE.EXE, RESTORE.EXE, SETBOOT.EXE, SORT.EXE, SPOOL.EXE, SYSLEVEL.EXE, TEDIT.EXE, TRACE.EXE, UNPACK.EXE, XCOPY.EXE, XDFCOPY.EXE, but also unfortunately TELNET and FTP, and some others; infact, almost any 16 bit application is a possible candidate. Note that CMD.EXE is an exception of the rule: it is largely still 16 bit, but is clean for XFree86/OS2 use.

You can use the EWS utility EXEMAP.EXE to check if these DLLs are linked in.

Note: the existance of these DLLs does not mean that the program does not work at all, actually EMX.DLL could call KBD API functions but usually does not in the X11 environment. However, certain unexpected effects may occur in an xterm. I'll try to find a workaround for this in the future, but don't count on this.

3.14

FTP seems to work in an xterm, but it does not hide the password?

This is a side effect of what was described in Q 3.13. So actually FTP does NOT work. [...]

II. Inter Process (Session) Communication

" 2.46

I found that killing the \mathcal{X} server or the window manager won't work well to end

X11. How can I shutdown X11?

Yes, we know it is a problem, mainly related to signal handling that is not fully compatible to Unix signaling as well as some differences in session semantics.

XFree86 is not really intended to be started up and shutdown all the time [...]

2.52

What does the error message "`_X11TransOs2OpenClient: Open server pipe PIPEXxf86.0 failed`" mean?

This is for the local named pipe communication the same problem as Q 2.31 is for the network connection. It basically means: the server crashed for some reason, and now a client, e.g. `xterm` cannot connect to the server. The reason is probably hidden somewhere else in the `XF86Config` file, or some other setup problem (e.g. network configuration). [...]

2.31

What does "`SocketINETConnect() can't connect: errno = 65`" mean?

What does "`SocketINETConnect() can't connect: errno = 61`" mean?

These messages **can** point out a network installation problem, in many cases they are secondary errors, though, and are really caused by a different problem.

Nevertheless, you should ensure that your network is setup correctly. [...]

3.8

How can I enable authorization for host `foo.edu` without using `xhost`?

Create the file `X0.hosts` in `XFree86libX11`. To enable connections from a certain host, add in a line containing

`inet:hostname` to the file. For example, your file may look like this:

inet:foo.edu

inet:friendly.host.edu

..

Note that the X0 refers to the zero in the display name

os2systemname:0, i.e. in the case of Q 3.6 this would

become a X1.hosts then. [...]

III. Emx and device driver programming (hardware devices)

Q: Can EMX be used to generate 16 bit code for programming OS/2 device drivers?

No!

:

>(Holger Veit): as gas lacks the ability to produce arbitrary segment

>types beyond 32 bit .text/.data/.bss segments. It can generate 16 bit

>instructions in a 32 bit segment (prefixed with 0x66/0x67), though.

On Wed, 13 Oct 1999 17:58:35 +0100, Csaba Raduly wrote:

>But you can write device drivers in 32-bit using some (already existing)

> thunking mechanism

>

>Check out /pub/os2/system/drivers/filesys/32drv170.zip on hobbes.

IBM distributes the device driver devel package (including compiler)

for free! You may want to look at

<http://rover.wiesbaden.netsurf.de/~meile/warpstock/wsp02/en/index.html>

for the details...

II. Preliminary steps

The following is meant to overcome some simple, nevertheless annoying problems and to make things work faster: Quick and simple fixes, some dirty stolen tricks.

1) RTF docs

- Read the README file! Or should it better be called DON'TREADME (psychology?)
- Read the INSTALL file!
- Read any additional info files from the program authors. Many problems occurring under special UN*X flavours have a similar counterpart on OS/2 (others haven't, but this shall be a simple guide :-)

Read the `/XFree86/lib/X11/config/README` file, for a short introduction to adapting Imake templates. The following may be useful, too.

2) Some additions to `/XFree86/lib/X11/config/os2.cf` and `os2.rules`

I tried to make life easier, so I've added some defines to the vendor specific imake template file: `/XFree86/lib/X11/config/os2.cf`.

Reasons for that:

- Never edit generated Makefiles by hand: Far too much work for larger projects and errors are to be expected. (But it is sometimes unavoidable).
- If things work and don't break other projects, they should be mailed to the respective lists in order to finally become standard:

But certainly, as SMiyata points out: As "far as possible, the changes must be

confined to Imakefiles and definition files bundled in the application.

The settings in `os2.cf` and `os2.rules` are shared by other applications, and hence may cause conflicts in them. Also, the changes to the template files are local to

your setting, thus, although it may help to build binary on your system, it is not

suitable for porting, unless they are reflected in the distribution of XFree86 itself.

There are indeed the cases that modifications must be made on template files coming

with XFree86 rather than Imakefile and definition files coming with the applications:

[those often have uncommon names like 'local.cf' or 'machinefile', etc.]

Akira Hatakeyama <akira@sra.co.jp> reported elsewhere that, in order to build Xvnc on OS/2, the lines

```
#define BuildXKB          YES
#define BuildLBX          YES
```

in `os2.cf` must be modified to

```
#ifndef BuildXKB
#define BuildXKB          YES
#endif
#ifndef BuildLBX
#define BuildLBX          YES
#endif
```

since, else, the settings in `vnc.def` is overwritten by those of `os2.cf`"

My proposed changes of `/XFree86/lib/X11/config/os2.cf`:

150 CHAPTER 11. SOME NOTES ON PORTABILITY AND CODING STYLE

```
#define ManSuffix 1X11
#define LibManSuffix 3X11
#define FileManSuffix 4X11
```

instead of

```
#define ManSuffix 1 [... etc.]
```

(This helps a lot, if you already installed or will install several man pages from
bsd, Linux, etc. for the same function or tool.

I'm sure this will happen!! Also this is for good reasons now standard
on most recent UN*X versions.)

Add the following (after: #define
HasStrcasecmp NO):

```
#define StrcasecmpDefines -Dstrcasecmp=strcmp
-Dstrncasecmp=strncmp [and probably: -DNEED_STRCASECMP]
```

(Just a hack, until emx has a working 'strcasecmp()' function interface).

Change

```
#define CplusplusCmd g++
```

to

```
#define CplusplusCmd gcc
#define CplusplusOptions -D__ST_MT_ERRNO__ -Zmtd -Zsysv-signals -
O2
```

(I can't find any reason to call emx gcc by a different name for C++
or to use different options here.)

b) EMX docs recommend using object module format (-Zomf; link386.exe) for better stability, if neither gdb.exe nor fork() shall be used. The following additions to /XFree86/lib/X11/config/os2.rules are meant to facilitate the switch in this case; nevertheless you'll have to check dependency lines and rules, if a.out format is still used (and some rules for libraries will become obsolete, but not harmful):

```
XCOMM $XConsortium: os2.rules /main/1 1996/10/31 14:47:27 kaleb $
XCOMM platform: $XFree86: xc/config/cf/os2.rules,v 3.15 1997/01/05
11:49:39 dawes Exp $
```

```
.SUFFIXES: .a .lib .o .obj .c .C .cc .cpp
```

```
.c.$O:
```

```
$(CC) $(CFLAGS) -c $*.c
```

```
#if HasCplusplus
```

```
.$C.$O:
```

```
$(CXX) $(CFLAGS) -c $*.$C
```

```
#endif
```

```
XCOMM Uncomment if using OMF (emxomf called automatically by gcc)!
```

```
XCOMM CC = gcc -Zomf
```

```
XCOMM O = obj
```

```
XCOMM LD = emxomfld
```

```
C = C XCOMM or .cpp
```

```
O = o
```

```
A = .a
```

III. Common initial problems

Initially I thought imake was a labyrinthic beast, deserving the title of the Minotaurus among the make tools. But with small and simple fixes to work around some annoying bugs I was able to tame it a bit, so that 'xmkmf' at least does something useful:

- 1) 'xmkmf -a' crashes because of 'sh' and 'Rexx' conflicts
-

Usually Imakefiles silently assume they are running on UNIX in a Bourne 'sh' environment, since authors intentionally do not support non UNIX platforms.

Just try running the following changed Rexx scripts first, to create a working environment:

```
make.cmd
```

```
/* REXX */
```

```
'@echo off'
```

```
PARSE ARG a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
```

```
/* x11make.exe 'MAKE=x11make.exe SHELL=d:/os2/cmd.exe' a1 a2 a3 a4 a5  
a6 a7 a8 a9 a10 */
```

```
'SET SHELL=D:OS2CMD.EXE'
```

```
'SET X11SHELL=D:OS2CMD.EXE'
```

```
'set emxshell=D:OS2CMD.EXE'
```

```
'set CONFIG_SHELL=D:OS2CMD.EXE'
```

```
'set MAKE_SHELL=D:OS2CMD.EXE'
```

```
x11make.exe 'MAKE=x11make.exe SHELL= MAKE_SHELL=d:os2cmd.exe  
CONFIG_SHELL=d:os2cmd.exe' a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
```

or

gmake.cmd

```
/* REXX */
```

```
'@echo off'
```

```
PARSE ARG a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
```

```
'set SHELL=d:/bin/ash.exe'
```

```
'SET X11SHELL=D:OS2CMD.EXE'
```

```
'set emxshell=d:/bin/ash.exe'
```

```
'set MAKE_SHELL=d:/bin/ash.exe'
```

```
/* x11make.exe 'MAKE=x11make.exe SHELL=' a1 a2 a3 a4 a5 a6 a7 a8 a9  
a10
```

```
*/
```

```
x11make.exe 'MAKE=x11make.exe SHELL=d:/bin/ash.exe
```

```
MAKE_SHELL=d:/bin/ash.exe CONFIG_SHELL=d:/bin/ash.exe' a1 a2 a3  
a4 a5
```

```
a6 a7 a8 a9 a10
```

```
=====
```

Normally, running make.cmd should be sufficient, since the current OS/2 implementation by H. Veit and others makes extensive use of Rexx, needs cmd.exe as command processor and is incompatible with Bourne shells. Here a number of problems arise with many common Imakefiles.

Now lets assume xmkmf worked. Theoretically the steps are:

```
xmkmf
```

```
make Makefiles
```

```

make depend
make all (or simply: make)
.programname.exe for testing
make install
make install.man

```

2) 'make' crashes because of 'sh' and 'Rexx' conflicts

Imake is misconfigured on most (commercial) UN*X platforms. Authors normally compensate this by including many lines of (normally highly complicated) Bourne shell code and calling several Bourne shell scripts in the Imakefile. In this case 'make' won't work. Nor does a sh.exe as make shell work: The current implementation depends on cmd.exe and Rexx. Finding and rewriting everything in those immense Imakefiles, helper scripts and Makefiles isn't really an alternative. Just try alternatively make and gmake (having installed e.g. /bin/ash.exe). Now make should simply resume where it has crashed. Just delete 'products' of those crashes (empty script output, truncated object files, core dumps, etc.) before resuming the process. This makes using Imake real fun!

Tips by John Williams <jwilliams@commerce.otago.ac.nz> (concerning 'configure' of GNU 'autoconf', fixed by the OS/2-port, but still valid for certain parallel Bourne sh-scripts very common in Imakefiles and/or config.local, etc.):

"3. Configure couldn't search the OS/2 path for programs because the shell construct:

```

for foo in bar do blah

```

apparently needs 'bar' to be a space or colon delimited list, whereas the OS/2 \$PATH is semi-colon delimited.

4. Configure could not check to see if it's test programs were being compiled because

it tests for success using test -s which means test that the file is not empty.

The emx port of GCC creates foo.exe executables and touches foo, leaving an empty

file. Why does autoconf test for success with a non-empty file (-s) and not a 'file exists' (-f) test?

5. Overcome the above problems by setting an environment variable UNIX_PATH to be

a colon delimited path. Next, search and replace using sed thus:

```
sed -f os2patch.sed configure > configure.os2
```

where os2patch.sed looks like:

```
s/PATH/UNIX_PATH/
s/test -s/test -f/
s/host=NONE/host=OS2/
s/"${IFS}:"$/${IFS};"/
```

and then execute sh Configure.os2. Lines three and four are due to Ilya Zakharevich,

who also pointed out the following issues:

1.if /tmp doesn't exist, make it

2.ensure CONFIG_SHELL=sh set in shell environment you are running Configure

from.

6. You can find what functions are contained in what object files and libraries by using 'nm' (comes with emx). This also tells you which functions are referenced (called) by functions in that file."

IV. Imakefile examples

\$XFree86: config/README states:

"The easiest way to write an Imakefile is to find another one that does something similar and copy/modify it!"

Here some strongly shortened examples that use Bourne shell code:

```
sh -c 'for i in "$@"; do ... etc., etc.
```

1) creating a static library (archive): NormalLibraryTarget(rw, \$(OBJS))

```

-----snip-----
# Imakefile for xpaint read/write directory

#include "../version"
#include "../Local.config"
INCLUDES = -I. $(XPM_INCLUDE) $(TIFF_INCLUDE) $(JPEG_INCLUDE)
$(PNG_INCLUDE)

# Nothing to change below here

TIFF_SRC = writeTIFF.c readTIFF.c
#ifdef HaveTIFF
TIFF_OBJ = writeTIFF.$O readTIFF.$O

```

```

TIFF_DEF = -DHAVE_TIFF
#endif
[...]
XPM_SRC = readWriteXPM.c
XPM_OBJ = readWriteXPM.$O

SGI_SRC = readWriteSGI.c
#ifdef SGIArchitecture
SGI_OBJ = readWriteSGI.$O
SGI_DEF = -DHAVE_SGI
#endif

DEFINES = $(XPM_DEF) $(TIFF_DEF) $(SGI_DEF) $(JPEG_DEF) $(PNG_DEF)

SRCS = rwTable.c
      readWriteXBM.c readWritePNM.c readWriteXWD.c writePS.c
      readGIF.c writeGIF.c $(XPM_SRC) $(TIFF_SRC) $(SGI_SRC)
      $(JPEG_SRC) $(PNG_SRC) libpnmrw.c
OBJS = rwTable.$O
      readWriteXBM.$O readWritePNM.$O readWriteXWD.$O writePS.$O
      readGIF.$O writeGIF.$O $(XPM_OBJ) $(TIFF_OBJ) $(SGI_OBJ)
      $(JPEG_OBJ) $(PNG_OBJ) libpnmrw.$O

HDRS = libpnmrw.h rwTable.h
MISC = Imakefile

NormalLibraryTarget(rw, $(OBJS))

makelist:
    sh -c 'for i in `$(TEXT) $(SRCS) $(MISC) $(HDRS)` ; do
        echo $(CURRENT_DIR)/$$i >> ../filelist ; done '

rwTable.$O: ../Local.config
    -----snap-----

```

D. What Shall I Better Change in the Sources?

Last but not least some well meant recommendations from Chapter 5 of the "emx APPLICATION DEVELOPER'S GUIDE", with some additional comments.

You might use them as a short checklist.

I. Globbing

" If you want Unix-like wildcard expansion built into the program, use

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

This should be done at the very beginning of main(), before ARGV and ARGV are used. See _wildcard() and _response()."

II. DOS "text" mode, binary reading, etc.

"Famous problems are related to the
binary/text mode mess on OS/2,
especially when using stdin/stdout."

(Alexander Mai)

" Programs using stdin, stdout or stderr for binary data should call `_fsetmode()` to switch the stream to binary mode."

" Change all `open()`, `fopen()`, `fdopen()` and `freopen()` calls to use `O_BINARY` or "b", respectively, for binary files. If a file contains both binary and textual data, read the file in binary mode and do the conversion yourself."

Note: XFree86/OS2 uses '-Zbin-files'. So "do the conversion yourself" "and ignore carriage returns" and <CTRL Z>.

" 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."

III. Process management and `fork()`

"Replace `fork()` and `exec*()` with `spawn*()`.

Under OS/2, `fork()` is inefficient."

(emx docs)

Note: This is usually not difficult; it is necessary to use '-Zomf'. Cf. the

example below.

Replacing vfork()-type lightweight processes with threads is complicated.

1) What is the difference between fork() and spawnvp()?

a) fork() and emx

The Posix syscall fork() is for traditional Unix systems the only possibility to create a new process i.e. a nearly identical clone of an existing one:

"This is achieved by copying the pointers in the process information block (PIB) of the parent to the newly allocated block in the kernel. On the other hand, on DOS

(Windows included) and on OS/2 (Windows NT included) there is no "fork" system call

but there is a "spawn" instead. This system call just allocates a new process information block. Only file handles and the environment variable space are inherited and selectors to the address space of the parent process are not copied. So the child cannot access the contents of the memory of the parent. In fact, for

OS/2 and Windows NT, it is impossible to implement "fork" in the same way as on the

UNIX, since even root has only read access to the PIB table in the kernel.

[In fact, there is no such security leak as 'root' in OS/2 (AH)]

EMX library function fork() is actually spawn system call with a little bit of IPC to copy memory contents. It is inefficient because all the memory contents rather than just pointers are copied.

Well, the difference I was talking was that a *spawned* child has a clean address

space, while a *forked* one might have a corrupted one." (SMiyata).

b) making it work

"fork() works only in programs linked by ld. It does not work in programs linked

by LINK386." [-Zomf]

"fork() doesn't work correctly in multithread programs."

"If the parent process uses the general terminal interface for the keyboard, the child process cannot read from the keyboard using `_read_kbd()`, the general terminal interface, or the Kbd OS/2 API functions.

If the process has a non-contiguous heap (that is, multiple heap objects), fork() will fail. Increase the initial heap size to work around this problem. If any DLL

used by the program shares the heap with the program and uses the heap in `_DLL_InitTerm()` by calling `sbrk()`, `malloc()`, etc., fork() will fail.

In both cases, fork() will return -1 and set `errno` to `ENOMEM`.

See also: `alarm()`, `exec*()`, `_rmutex_create()`, `sbrk()`, `sigaction()`, `signal()`, `spawn*()`, `_uflags()`, `wait()`, `waitpid()`"

See the emx reference for more details about caveats. Use huge stack and heap size.

c) replacing fork() and `execvp()`, etc.

Excerpt from `exec*()` documentation:

" When the new process ends, the parent process will be notified by `SIGCHLD`

and `wait()` will return the original process ID (PID). [...]

Restrictions:

Native DOS programs cannot be run. The new process gets a new process ID.

Unless the new process is an `emx` program (using `emx.dll` or `emx.exe`), the actions for all signals are reset to `SIG_DFL` in the new process"

"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 `SIGCLD`). 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."

What does this mean? Well, in a Posix OS traditionally the parent process clones

itself with `fork()`, which generates a new process environment, whereas `exec()` overwrites its own present environment with a new one, keeping the PID.

Usually this is combined:

`exec()` is used in the forked child process, overwriting the parent's process status with a new one. Meanwhile the parent usually calls `wait()` or a home-brew function and waits for child termination, communicated by `SIGCHLD`. `ps` (`pstat.exe`) gives you the details.

A common problem here is, that the child after termination does not find this `wait()` etc. point of the parent. Thus a 'zombie' remains. This may lead to a process table overflow, so that the system has to be shutdown and rebooted. All this hassle can be simplified by using `spawn*()`.

But for tracing and debugging:

"Do not use the PTRACE_TRACEME request of ptrace(): use P_DEBUG instead when starting the process with spawn*()."

IV. Unix file system issues

The separator of path components is a slash '/' as opposed to OS/2's ' '.

Note that many APIs understand the '/' on OS/2, but cmd.exe and most apps don't.

Multiple paths (e.g. in environment variables like PATH) are separated by a colon ':'

The root dir on every un*x system is '/'. Often code checks whether a given path is an

absolute one by comparing the beginning with '/'. Use grep for all standard paths and

all commands which are mentioned in this introduction. (_fullpath() and _abspath()

can also be useful.)

Using

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

may help to support drive letters. Read the EMX docs:

"Watch out for Unix file system hacks: Unix allows deleting and renaming an open file (the file will be deleted after being closed).

[unlink(), remove(), rename()]

- Watch out for Unix file names (Unix is case sensitive, long file names and multiple dots are allowed). On OS/2's HPFS multiple dots are also allowed; however, trailing dots are not significant (except for the special file names '.' and '..').

- 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"`, respectively. However,

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

won't work as the standard OS/2 and DOS command interpreters don't recognize `/dev/null`.

- If you want to use for separating directories, changes may be necessary. These changes are optional because `/` also works.
- Implement support for drive names. This can be done by using

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

In addition, some changes will be necessary. For instance, you have to change code which checks whether a filename is an absolute path name. `_fullpath()` and `_abspath()` can also be useful."

"Unix does not have drive letters, so a file (resources, fonts, config, etc.) path starts with `'/'` normally. XFree86 is located in a tree below `%X11ROOT%XFree86...`

To prepend the X11ROOT part to a file path, a special function named `__XOS2RedirRoot` exists which you are supposed to use in these cases rather than an insane own solution. The prototype of this function is:

```
const char * __XOS2RedirRoot(const char *pathname);'
```

[...] XFree86/OS2 stores all configuration files in a common directory. This is `X11ROOT:XFree86libX11`, where X11ROOT is an environment variable which contains the drive letter the directory tree is located. The common places like `/usr/X11`,

`/usr/X11R6`, `/usr/XFree86`, `/etc` which are used for XFree86 in various Unix flavors, are not supported [...].

It is also there where you should put your XF86Config file, the color database (`rgb.txt`), the host authorization files (`X0.hosts`), and various other files." (FAQ)

"Potential problem might be the DLL name: IIRC one can either use

`'foo'` or `'x:barfoo.dll'`

to access `foo.dll` (if in `LIBPATH`)."
(Alexander Mai)

"Note that `///abc` is a valid Unix filename. It's equivalent to `/abc`.

- Note that `chdir("..")` is a no-op under Unix if the current working directory is the root directory. Under `emx`, `chdir("..")` fails in the root directory.
- Use `termio` or `termios` or read the keyboard with `_read_kbd()` if you don't want to get input line by line.
- Under Unix, directories in environment variables (`PATH`, for instance) are separated by colons; use semicolons instead.
- The shell isn't called `/bin/sh`. Use `system()`. `system()` and `popen()` don't expand wildcards (unless `COMSPEC` points to a shell which expands wildcards)."

So, in short:

"grep for `'/dev'`, `'/bin'`, `'/tmp'`, `'/usr'`, `'/proc'`, ...

also check out all commands starting up another process:

`exec*`, `popen`, `system`,...

Remember that `cmd.exe` doesn't like slashes!" (Alexander Mai)