The Linux ELF HOWTO

Daniel Barlow <daniel.barlow@sjc.ox.ac.uk> v1.11, 13 September 1995

This document describes how to migrate your Linux system to compile and run programs in the ELF binary format. It falls into three conceptual parts: (1) What ELF is, and why/whether you should upgrade, (2) How to upgrade to ELF-capability, and (3) what you can do then.

1 What is ELF? An introduction

ELF (Executable and Linking Format) is a binary format originally developed by USL (UNIX System Laboratories) and currently used in Solaris and System V Release 4. Because of its increased flexibility over the older a.out format that Linux currently uses, the GCC and C library developers decided last year to move to using ELF as the Linux standard binary format also.

This 'increased flexibility' manifests as essentially two benefits to the average applications progammer:

- It is much simpler to make shared libraries with ELF. Typically, just compile all the object files with `-fPIC`, then link with a command like

  ```
  gcc -shared -Wl,-soname,libfoo.so.y -o libfoo.so.y.x *.o
  ```

  If that looks complex, you obviously haven't ever read up on the equivalent procedure for a.out shared libraries, which involves compiling the library twice, reserving space for all the data you think that the library is likely to require in future, and registering that address space with a third party (it's described in a document over 20 pages long — look at `<ftp://tsx-11.mit.edu/pub/linux/packages/GCC/src/tools-2.16.tar.gz>` for details).

- It makes dynamic loading (ie programs which can load modules at runtime) much simpler. This is used by Perl 5, Python, and the ongoing port of Java to Linux, among other things. Other suggestions for dynamic loading have included super-fast MUDs, where extra code could be compiled and linked into the running executable without having to stop and restart the program.

Against this it must be weighed that ELF is possibly a bit slower. The figures that get bandied around are between 1% and 5%, though all the actual tests that have been conducted so far indicate that the difference is small enough to get lost in the noise of other events happening at the same time. If you have TeX or a Postscript viewer/printer, you can read `speed.comp-1.0.tar.gz`, which is available from SunSite somewhere.

The slowdown comes from the fact that ELF library code must be position independent (this is what the `-fPIC` above stands for) and so a register must be devoted to holding offsets. That's one less for holding variables in, and the 80x86 has a paucity of general-purpose registers anyway.

1.1 What ELF isn't

There are a number of common misconceptions about what ELF will do for your system:

It's not a way to run SVR4 or Solaris programs

Although it's the same binary 'container' as SVR4 systems use, that doesn't mean that SVR4 programs suddenly become runnable on Linux. It's analogous to a disk format — you can keep Linux programs on MSDOS or Minix-format disks, and vice versa, but that doesn't mean that these systems become able to run each others' programs.

It is theoretically possible to run applications for other x86 Unices under Linux, but following the instructions in this HOWTO will *not* have that effect. Start by looking at the iBCS kernel module (somewhere on `tsx-11.mit.edu`) and see if it fits your needs.

It's not intrinsically smaller or faster

You may well end up with smaller binaries anyway, though, as you can more easily create shared libraries of common code between many programs. In general, if you use the same compiler options and your binaries come out smaller than they did with a.out, it's more likely to be fluke or a different compiler version. As for 'faster', I'd be surprised. Speed increases could turn up if your binaries are smaller, due to less swapping or larger functional areas fitting in cache.

It doesn't require that you replace every binary on your system

At the end of this procedure you have a system capable of compiling and running both ELF and a.out programs. New programs will by default be compiled in ELF, though this can be overridden with a command-line switch. There is admittedly a memory penalty for running a mixed ELF/a.out system — if you have both breeds of program running at once you also have two copies of the C library in core, and so on. I've had reports that the speed difference from this is undetectable in normal use on a 6Mb system though (I certainly haven't noticed much in 8Mb), so it's hardly pressing. You lose far more memory every day by running bloated programs like Emacs and static Mosaic/Netscape binaries :-)

It's nothing to do with Tolkien.

Or at least, not in this context.

1.2 Why you should(n't) convert to ELF

There are essentially two reasons to upgrade your system to compile and run ELF programs: the first is the increased flexibility in programming referred to above, and the second is that, due to the first, everyone else will be too. Future releases of the C library and GCC will only be compiled for ELF, and other developers are expected to move ELFwards too.

Pleasingly for the purposes of symmetry, there are also two reasons not to convert at this time. The first is that things are still changing, some packages (including the 'stable' 1.2 kernel series) require patches to be made before they will compile in ELF, and there may be residual bugs; one could make a strong case for waiting until Linus himself has converted, for example.

The second is that although the installation described here is a fairly small job in itself (it can be completed in well under an hour, excepting the time taken to download the new software), an error at almost any stage of it will probably leave you with an unbootable system. If you are not comfortable with upgrading shared libraries and the commands `ldconfig` and `ldd` mean nothing to you, you may want to obtain or wait for a new Linux distribution in ELF, and backup, reinstall and selectively restore your system using it. Then again (and especially if the system is not mission-critical) you may want to go through it anyway and look on it as a learning experience.

Still with us?

2 Installation

2.1 Background

The aim of this conversion is to leave you with a system which can build and run both a.out and ELF programs, with each type of program being able to find its appropriate breed of shared libraries. This obviously requires a bit more intelligence in the library search routines than the simple 'look in `/lib`, `/usr/lib` and anywhere else that the program was compiled to search' strategy that some other systems can get away with.

The beastie responsible for searching out libraries in linux is `/lib/ld.so`. The compiler and linker do not encode absolute library pathnames into the programs they output; instead they put the library name and the absolute path to `ld.so` in, and leave `ld.so` to match the library name to the appropriate place at runtime. This has one very important effect — it means that the libraries that a program uses can be moved to other directories *without recompiling the program*, provided that `ld.so` is told to search the new directory. This is essential functionality for the directory swapping operation that follows.

The corollary of the above, of course, is that any attempt to delete or move `ld.so` will cause *every dynamically linked program on the system to stop working*. This is generally regarded as a Bad Thing.

For ELF binaries, an alternate dynamic loader is provided. This is `/lib/ld-linux.so.1`, and does exactly the same thing as `ld.so`, but for ELF programs. `ld-linux.so.1` uses the same support files and programs (`ldd`, `ldconfig`, and `/etc/ld.so.conf`) as the a.out loader `ld.so` does.

The basic plan, then, is that ELF development things (compilers, include files and libraries) go into `/usr/{bin,lib,include}` where your a.out ones currently are, and the a.out things will be moved into `/usr/i486-linuxaout/{bin, lib, include}`. `/etc/ld.so.conf` lists all the places on the system where libraries are expected to be found, and `ldconfig` is intelligent enough to distinguish between ELF and a.out variants.

There are a couple of exceptions to the library placement, though.

- Some old programs were built without the use of `ld.so`. These would all cease working if their libraries were moved from under them. Thus, `libc.so*` and `libm.so*` must stay where they are in `/lib`, and the ELF versions have had their major numbers upgraded so that they do not overwrite the a.out ones. Old X libraries (prior to version 6) are best left where they are also, although newer ones (`libX*so.6`)

must be moved. Moving the old ones will apparently break xview programs, and not moving the new ones will cause them to be overwritten when you install ELF X libraries.

If you have non-ld.so programs that require libraries other than the above (if you know which programs they are, you can run ldd on them to find out which libraries they need *before* breaking them) you have essentially two options. One, you can extract the ELF library tar files into a temporary directory, check whether your precious library would be overwritten, and if so, move the ELF version of the library into, say, `/usr/i486-linux/lib` instead of `/lib`. Make sure your `ld.so.conf` has `/usr/i486-linux/lib` in it, then run `ldconfig` and think no more on't. Two, you can recompile or acquire a newer copy of the offending program. This might not be a bad idea, if possible.

- If you have `/usr` and `/` on different partitions, you'll need to move at least some of the libraries in `/lib` to somewhere on the root disk, not on `/usr`. Either you can go through the programs that you need to run at system startup or when in single-user mode, and identify the libraries they use, or you can depend on your system/distribution integrator to have done this for you and just move all (er ... some. See above for the exceptions) of the libraries in `/lib` to `/lib-aout`.

2.2 Before you start — Notes and Caveats

- You will need to be running a post-1.1.52 **kernel with ELF binary format support**.
- You are recommended to prepare or acquire a linux boot/root disk, such as a Slackware rescue disk. You probably won't need it, but if you do and you don't have one, you'll kick yourself. In a similar 'prevention is better than cure' vein, statically linked copies of `mv`, `ln`, and maybe other file manipulation commands (though in fact I think you can do everything else you actually *need* to with shell builtins) may help you out of any awkward situations you could end up in.
- If you have been following the ELF development, you may have ELF libraries in `/lib/elf` (usually `libc.so.4` and co). Applications that you built using these should be rebuilt, then the directory removed. There is no need for a `/lib/elf` directory!
- Most Linux installations these days have converged on the 'FSSTND' standard file system, but doubtless there are still installed systems that haven't. If you see references to `/sbin/`*something* and you don't have a `/sbin` directory, you'll probably find the program referred to in `/bin` or `/etc/`.

2.3 You will need ...

The following packages are available from `<ftp://tsx-11.mit.edu/pub/linux/packages/GCC/>` and `<ftp://sunsite.unc.edu/pub/Linux/GCC/>`. Both sites are widely mirrored; please take the time to look up your nearest mirror site and use that instead of the master sites where possible. It's faster for both you and everyone else.

These packages (either the listed version or a later one) are required. Also download and read through the release notes for each of them: these are the files named `release.`*packagename*. This applies especially if you get newer versions than are listed here, as procedures may have changed.

- `ld.so-1.7.3.tar.gz` — the new dynamic linker

- `libc-5.0.9.bin.tar.gz` — the ELF shared images for the C library and its friends (`m` (maths), `termcap`, `gdbm`, and so on), plus the corresponding static libraries and the include files needed to compile programs with them. libc 5.2.something is expected to be released during the lifetime of this HOWTO, and is considerably different from 5.0.9; if you want to install it, you're on your own, but I'd recommend installing 5.0.9 first and then installing it over the top. There are several parts to libc 5.0.9 which are not included in 5.2.x and for which the distribution channels are not entirely set up yet.
- `gcc-2.7.0.bin.tar.gz` — the ELF C compiler. Also includes an a.out C compiler which understands the new directory layout.
- `binutils-2.5.21.17.bin.tar.gz` — the GNU binary utilities patched for Linux. These are programs such as `gas`, `ld`, `strings` and so on, most of which are required to make the C compiler go.

2.4 Rearranging your filesystem

Sooo... Note that in all that follows, when I say 'remove' I naturally mean 'backup then remove' :-). Also, these instructions directly apply only to people who haven't yet messed with ELF — those who have are expected to have the necessary nous to adapt as appropriate. Let's go!

1. Make the new directories that you will move a.out things to

```
mkdir -p /usr/i486-linuxaout/bin
mkdir -p /usr/i486-linuxaout/include
mkdir -p /usr/i486-linuxaout/lib
mkdir /lib-aout
```

2. Untar the dynamic linker package `ld.so-1.7.3` in the directory you usually put source code, then read through the `ld.so-1.7.3/instldso.sh` script just unpacked. If you have a really standard system, run it by doing `sh instldso.sh`, but if you have anything at all unusual then do the install by hand instead. 'Anything at all unusual' includes
   - using zsh as a shell (some versions of zsh define `$VERSION`, which seems to confuse `instldso.sh`)
   - having symlinks from `/lib/elf` to `/lib` (which you shouldn't need, but you may have valid reasons for if you have been following the ELF development)
3. Edit `/etc/ld.so.conf` to add the new directory `/usr/i486-linuxaout/lib` (and `/lib-aout` if you're going to need one). Then rerun `/sbin/ldconfig -v` to check that it is picking up the new directories.
4. Move all the a.out libraries in `/usr/*/lib` to `/usr/i486-linuxaout/lib`. Note, I said 'libraries' not 'everything'. That's files matching the specification `lib*.so*` , `lib*.sa*`, or `lib*.a`. Don't start moving `/usr/lib/gcc-lib` or anything silly like that around.
5. Now look at `/lib`. Leave intact `libc.so*`, `libm.so*`, and `libdl.so*`. If you have symlinks to X libraries (`libX*.so.3*`) leave them there too — XView and some other packages may require them. Leave `ld.so*`, `ld-linux.so*` and any other files starting with `ld`. As for the remaining libraries (if you have any left): if you have `/usr` on the root partition, put them in `/usr/i486-linuxaout/lib`. If you have `/usr` mounted separately, put them in `/lib-aout`. Now run `ldconfig -v`

6. Remove the directory `/usr/lib/ldscripts` if it's there, in preparation for installing the binutils (which will recreate it)

7. Remove any copies of `ld` and `as` (*except* for `ld86` and `as86`) that you can find in `/usr/bin`.

8. Some versions of GNU tar appear to have problems dealing with symbolic links in the destination directory. You have two options here:

   (a) (preferred) Use `cpio` instead of `tar`, it doesn't have this problem. `zcat /wherever/you/put/it/libc-5.0.9.tar.gz | cpio -iv` is the magic incantation here, to be executed from the root directory.

   (b) (if you don't have cpio installed) Before installing the libc images you might want to go through `/usr/include` and remove some parts.

   This is icky. Many packages (such as ncurses) are installed into `/usr/include` by distribution maintainers and are *not* supplied with the C library. Backup the `/usr/include` tree, use `tar tzf` to see what's in the archive before untarring it, then delete the bits of `/usr/include` that it actually fills. Then untar the `libc-5.0.9.bin.tar.gz` package from root.

9. Install the binutils package. `tar -xvzf binutils-2.5.2.l17.bin.tar.gz -C /` is one perfectly good way to do this.

10. You have now installed everything you need to run ELF executables. Medical experts recommend that VDU workers take regular breaks away from the screen; this would be an opportune moment. Don't forget what you were doing, though; depending on the version of gcc you were previously using, you may have left yourself unable to compile programs in a.out until you unpack the new gcc.

11. Backup and remove everything in `/usr/lib/gcc-lib/{i486-linux, i486-linuxelf, i486-linuxaout}/` If you use a non-standard `gcc` driver (eg if you use Gnu ADA), copy that somewhere safe also. Then install the gcc package, again by untarring from root.

12. Some programs (notably various X programs) use `/lib/cpp`, which under Linux is generally a link to `/usr/lib/gcc-lib/i486-linux/`*version*`/cpp`. As the preceding step wiped out whatever version of `cpp` it was pointing to, you'll need to recreate the link:

```
$ cd /lib
$ ln -s /usr/lib/gcc-lib/i486-linux/2.7.0/cpp .
```

13. The FSSTND people have once again justified their keep by moving the `utmp` and `wtmp` files from `/var/adm` to `/var/run` and `/var/log` respectively. You'll need to add some links dependent on where they currently live, and you may need to make the `/var/log` and `/var/adm` directories too. I reproduce below the `ls -l` output of appropriate bits on my system:

```
$ ls -ld /var/adm /var/log /var/run /var/log/*tmp /var/run/*tmp
lrwxrwxrwx   1 root     root            3 May 24 05:53 /var/adm -> log/
drwxr-xr-x   9 root     root         1024 Aug 13 23:17 /var/log/
lrwxrwxrwx   1 root     root           11 Aug 13 23:17 /var/log/utmp -> ../run/utmp
-rw-r--r--   1 root     root       451472 Aug 13 23:00 /var/log/wtmp
drwxr-xr-x   2 root     root         1024 Aug 13 23:17 /var/run/
-rw-r--r--   1 root     root          448 Aug 13 23:00 /var/run/utmp
```

Check
the FSSTND (from LDP archives such as `<ftp://sunsite.unc.edu/pub/Linux/docs/fsstnd/>`) for the full story.

14. This step is optional. If you're intending to continue compiling programs in a.out, this is the appropriate time to install libc.so 4.7.*x*. Untar it from root, as you are now no doubt fully capable of doing without further explanation.

Done! Simple tests that you can try are

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.0/specs
gcc version 2.7.0
$ gcc -v -b i486-linuxaout
Reading specs from /usr/lib/gcc-lib/i486-linuxaout/2.7.0/specs
gcc version 2.7.0
$ ld -V
ld version cygnus/linux-2.5.21.14 (with BFD cygnus/linux-2.5.21.11)
  Supported emulations:
   elf_i386
   i386linux
   i386coff
```

followed of course by the traditional "Hello, world" program. Try it with `gcc` and with `gcc -b i486-linuxaout` to check that both the a.out and ELF compilers are set up corectly.

2.5 What it should look like (outline directory structure)

This is a deliberately vague guide to what the files you have just installed are. It may be useful for troubleshooting or deciding what to delete.

2.5.1 `/lib`

- Dynamic linkers `ld.so` (a.out) and `ld-linux.so.1` (ELF). Either of these may be symlinks, but make sure that the files they point to do exist.
- Basic shared libraries `libc.so.4`, `libm.so.4` (a.out) These are symlinks, but check that they point to real files.
- Basic shared libraries `libc.so.5`, `libm.so.5`, `libdl.so.1`,`libcurses.so.1`,`libtermcap.so.2`, (ELF). Again, these are symlinks.
- *Lots* of symlinks. For each library, there should be an actual file (for example `libc.so.5.0.9`), a symlink to it with only the major version number in its name (`libc.so.5`) and a symlink pointing to *that* with no version number (`libc.so`). That's

```
lrwxrwxrwx   1 root  root      9 May 24 05:52 libc.so -> libc.so.5
lrwxrwxrwx   1 root  root     13 Aug 25 12:48 libc.so.5 -> libc.so.5.0.9
-rwxr-xr-x   1 bin   bin  562683 May 19 04:47 libc.so.5.0.9
```

2.5.2 /usr/lib

- All the non-library files and directories that were there previously.
- libbfd.so*,libdb.so*, libgdbm.so*, ELF shared libraries. All to consist of three files as explained above in the /lib section.
- libbsd.a, libgmon.a, libldso.a, libmcheck.a, libieee.a, libmcheck.a and one lib*.a file for every ELF shared library in /lib and /usr/lib . ELF static libraries. The ones duplicating the shared libraries may not be tremendously useful for most people — when using ELF, you can use the gcc -g switch with shared libraries, so there's not much reason to compile static any longer.
- crt0.o, gcrt0.o. a.out 'start of program' files; one of these is linked as the first file in every a.out program you compile, unless you take steps to avoid it.
- crt1.o, crtbegin.o, crtbeginS.o, crtend.o, crtendS.o, crti.o, crtn.o, gcrt1.o. ELF startup files. These do similar things to *crt0.o above for ELF programs.

2.5.3 /usr/lib/ldscripts

- This is where the driver scripts for ld live, as the name suggests. It should look like

```
$ ls /usr/lib/ldscripts/
elf_i386.x      elf_i386.xs     i386coff.xn     i386linux.xbn
elf_i386.xbn    elf_i386.xu     i386coff.xr     i386linux.xn
elf_i386.xn     i386coff.x      i386coff.xu     i386linux.xr
elf_i386.xr     i386coff.xbn    i386linux.x     i386linux.xu
```

2.5.4 /usr/i486-linux/bin

- ar, as, gasp, ld, nm, ranlib, strip. These are all actually symlinks to the real binutils in /usr/bin

2.5.5 /usr/i486-linuxaout/bin

- as — the a.out assembler, and gasp, its macro preprocessor
- ar, ld, nm, ranlib, strip — symlinks to the real binutils in /usr/bin

2.5.6 /usr/i486-linux/lib

- ldscripts is a symlink to /usr/lib/ldscripts.

2.5.7 /usr/i486-linuxaout/lib

- lib*.so*. a.out shared library images. Needed to run a.out programs
- lib*.sa. a.out shared library stubs. Needed to compile a.out programs that use shared libraries. If you don't intend to, you can safely remove these.

- `lib*.a`. a.out static libraries. Needed to compile static a.out programs (eg when compiling with `-g`). Again, you can delete them if you don't intend to.
- `ldscripts` is a symbolic link to `/usr/lib/ldscripts`

2.5.8 `/usr/lib/gcc-lib/i486-linux/2.7.0`

- This directory contains a version of gcc 2.7.0 set up to compile ELF programs.

2.5.9 `/usr/lib/gcc-lib/i486-linuxaout/2.7.0`

- This directory contains a version of gcc 2.7.0 set up to compile a.out programs, which knows about the new directory structure. If you're not going to compile anything in a.out, deleting this may free up around 4Mb.

2.6 Common errors (Don't Panic!)

... in large friendly letters.

You moved the wrong thing and now nothing runs

You still have a shell running, though, and with a little ingenuity you can do an awful lot with shell builtins. Remember that `echo *` is an acceptable substitute for `ls`, and `echo >>filename` can be used to add lines to a file. Also, don't forget that `ldconfig` is linked static. If you moved, say, `libc.so.4` to `/lib-aout` mistakenly, you can do `echo "lib-aout" >>`/etc/ld.so.conf ; ldconfig -v/ and be back up again. If you moved `/lib/ld.so` you may be able to do `sln /silly/place/ld.so /lib/ld.so`, if you have a statically linked ln, and probably be back up again.

`no such file or directory: /usr/bin/gcc`

... when you know there *is* such a file. This usually means that the ELF dynamic loader `/lib/ld-linux.so.1` is not installed, or is unreadable for some reason. You should have installed it at around step 2 previously.

`not a ZMAGIC file, skipping`

from `ldconfig`. You have an old version of the ld.so package, so get a recent one. Again, see step 2 of the installation.

`bad address`

on attempting to run anything ELF. You're using kernel 1.3.x, where x<3. Upgrade to 1.3.3 or downgrade to 1.2.something

`_setutent: Can't open utmp file`

This message is often seen in multiples of three when you start an xterm. Go and read the FSSTND tirade near the end of the installation procedure.

`gcc: installation problem, cannot exec` *something*`: No such file or directory`

when attempting to do a.out compilations (*something* is usually one of `cpp` or `cc1`). Either it's right, or alternatively you typed

```
$ gcc -b -i486-linuxaout
```

when you should have typed

```
$ gcc -b i486-linuxaout
```

Note that the 'i486' does *not* start with a dash.

3 Building programs in ELF

3.1 Ordinary programs

To build a program in ELF, use `gcc` as always. To build in a.out, use `gcc -b i486-linuxaout` .

```
$ cat >hello.c
main() { printf("hello, world\n"); }
^D
$ gcc -o hello hello.c
$ file hello
hello: ELF 32-bit LSB executable i386 (386 and up) Version 1
$ ./hello
hello, world
```

This is perhaps an appropriate time to answer the question "if a.out compilers default to producing a program called `a.out`, what name does an ELF compiler give its output?". Still `a.out`, is the answer. Boring boring boring ... `:-)`

3.2 Building libraries

To build libfoo.so as a shared library, the basic steps look like this:

```
$ gcc -fPIC -c *.c
$ gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1.0 *.o
$ ln -s libfoo.so.1.0 libfoo.so.1
$ ln -s libfoo.so.1 libfoo.so
$ export LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH
```

This will generate a shared library called `libfoo.so.1.0`, and the appropriate links for ld (`libfoo.so`) and the dynamic linker (`libfoo.so.1`) to find it. To test, we add the current directory to `LD_LIBRARY_PATH`.

When you're happpy that the library works, you'll have to move it to, say, `/usr/local/lib`, and recreate the appropriate links. Note that the `libfoo.so` link should point to `libfoo.so.1`, so it doesn't need updating on every minor version number change. The link from `libfoo.so.1` to `libfoo.so.1.0` is kept up to date by `ldconfig`, which on most systems is run as part of the boot process.

```
$ su
# cp libfoo.so.1.0 /usr/local/lib
# /sbin/ldconfig
# ( cd /usr/local/lib ; ln -s libfoo.so.1 libfoo.so )
```

3.3 Programs with dynamic loading

These are covered extensively in H J Lu's ELF programming document, and the `dlopen(3)` manual page, which can be found in the ld.so package. Here's a nice simple example though: link it with `-ldl`

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

main()
{
  void *libc;
  void (*printf_call)();

  if(libc=dlopen("/lib/libc.so.5",RTLD_LAZY))
  {
    printf_call=dlsym(libc,"printf");
    (*printf_call)("hello, world\n");
  }

}
```

3.4 Debugging

Your existing copy of `gdb` will most likely work unchanged with ELF programs. The new version in the `GCC` directory on tsx-11 is reported to be better at debugging programs that use shared libraries and dynamic loading, and to understand ELF core dumps.

Note that 1.2 series kernels cannot generate core dumps from ELF programs anyway. 1.3 can.

4 Patches and binaries

At this point in the proceedings, you can, if you like, stop. You have installed everything necessary to compile and run ELF programs.

You may wish to rebuild some programs in ELF, either for purposes of 'neatness' or to minimise memory usage. For most end-user applications, this is pretty simple; some packages however do assume too much about the systems they run on, and may fail due to one or more of:

- Different underscore conventions in the assembler: in an a.out executable, external labels get `_` prefixed to them; in an ELF executable, they don't. This makes no difference until you start integrating hand-written assembler: all the labels of the form `_foo` must be translated to `foo`, or (if you want to be

portable about it) to `EXTERNAL(foo)` where `EXTERNAL` is some macro which returns either its argument (if `__ELF__` is defined) or `_` concatenated with its argument if not.

- Differences in libc 5 from libc 4. The interface to the locale support has changed, for one.
- The application or build process depending on knowledge of the binary format used — emacs, for example, dumps its memory image to disk in executable format, so obviously needs to know what format your executables are in.
- The application consists of or includes shared libraries (X11 is the obvious example). These will obviously need changes to accomodate the different method of shared library creation in ELF.

Anyway, here are two lists: the first is of programs that needed changing for ELF where the changes have been made (ie that you will need new versions of to compile as ELF), and the second is of programs that still need third-party patches of some kind.

4.1 Upgrade:

- **Dosemu**. Modulo the three or four cuurrent dosemu development trees (don't ask, just join the linux-msdos mailing list), dosemu runs with ELF. You'll need to monkey with the Makefile. Current versions of dosemu are available from `<ftp://tsx-11.mit.edu/pub/linux/ALPHA/dosemu/>`
- **Emacs**. Emacs has a rather odd build procedure that involves running a minimal version of itself, loading in all the useful bits as lisp, then dumping its memory image back to disk as an executable file. (FSF) Emacs 19.29 and XEmacs 19.12 (formerly Lucid Emacs) can both detect whether you are compiling as ELF and Do The Right Thing automatically.
- **MAKEDEV**. In some incarnations, this utility removes existing entries for devices before recreating them. This is Bad News if it happens to touch `/dev/zero`, as said device is necessary to the operation of all ELF programs. See the util-linux package(q.v.) for a fixed version.
- **perl 5.001**. Perl 5.001 plus the "official unofficial" patches a-e will compile unchanged on an ELF system, complete with dynamic loading. The patches are available from `ftp.metronet.com` or `ftp.wpi.edu`
- The `cal` program in **util-linux 2.2** doesn't work. Upgrade to *version 2.4* `<ftp://tsx-11.mit.edu/pub/linux/packages/utils>` or later.
- **XFree86**. XFree86 3.1.2 comes in both a.out and ELF formats. `ftp` to `ftp.xfree86.org`, read the 'too many users' message that you are almost guaranteed to get, and pick the closest mirror site network-wise to you. Once you have the contents of the `common` and `elf` directories, you must edit `/usr/X11R6/lib/X11/config/linux.cf` to change the lines saying

```
#define LinuxElfDefault         NO
#define UseElfFormat            NO
```

to say `YES` instead. Otherwise an xpm build will attempt to do odd stuff with `jumpas` and its associated relics of the past.

- **Mosaic**. I don't have the facilities to build this myself, but the Mosaic 2.7b1 binary available from NCSA comes in ELF. It has been linked against an odd X setup though, with the result that on normal systems it will complain about not finding `libXpm.so.4.5`. The simple fix is to edit it carefully with emacs or another editor that copes with binary files. Find the occurence of the string `libXpm.so.4.5^@` (where `^@` is a NUL — ASCII zero — character), delete the `.5` and add two more characters after the NUL to aviod changing the file length.

4.2 Patch

- **e2fsutils**. The Utilities for the Second Extended File System need a patch from `<ftp://ftp.ibp.fr/pub/linux/ELF/patches/e2fsprogs-0.5b.elf.diff.gz>` to compile correctly as a shared library. Remy Card says "This is the ELF patch which will probably be included in the next release of e2fsck and co"

- **file**. This works anyway, but can be improved: `<http://sable.ox.ac.uk/ jo95004/patches/file.diff>` adds support for identifying stripped and mixed-endian ELF binaries.

- **The Kernel**. As from at least 1.3.8, the development 1.3 series have a `make config` option to build using ELF tools. If you are using the 1.2 series, you have two options:

  1. Patch the Makefile slightly to use the a.out compiler. Just change the `CC` and `LD` definitions to be

```
LD      =ld -m i386linux
CC      =gcc -b i486-linuxaout -D__KERNEL__ -I$(TOPDIR)/include
```

  Alternatively,

  2. Apply H J Lu's patch which allows compiling the kernel in ELF (and also adds the ability to do ELF core dumps).

  Let me reiterate that neither of these is necessary for the 1.3 series.

- **ps (procps-0.97)** The `psupdate` program needs a patch to work if you have compiled the kernel as ELF. It's available in `<linux.nrao.edu:/pub/people/juphoff/procps>`, both as a patch to vanilla 0.97 and as an entire tar-file. A new version of procps is expected to be released soon with this patch in place, so if you can find procps 0.98 by the time you read this, this patch will probably be obsolete.

5 Further information

- The `linux-gcc` mailing list is really the best place to see what's happening, usually without even posting to it. Remember, it's not Usenet, so keep the questions down unless you're actually developing. For instructions on joining the mailing list, mail a message containing the word `help` to `majordomo@vger.rutgers.edu`. Archives of the list are at `<http://homer.ncm.com/>`.

- There's a certain amount of information about what the linux-gcc list is doing at my *ELF web page* `<http://sable.ox.ac.uk/ jo95004/elf.html>`, when I remember to update it. This also has a link to the latest version of this HOWTO, and the patches it refers to. For US people and others with poor

links to UK academic sites (that's nearly everyone outside of UK academia), this is all mirrored at `<http://www.blackdown.org/elf/elf.html>`

- See also Bobby Shmit's *ELF upgrade experience* `<http://www.intac.com/ cully/elf.html>` web page.
- The *GCC-FAQ* `<ftp://sunsite.unc.edu/pub/Linux/docs/faqs/GCC-FAQ.html>` contains much general development information and some more technical ELF details.
- There's also documentation for the file format on *tsx-11* `<ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz>`. This is probably of most use to people who want to understand, debug or rewrite programs that deal directly with binary objects.
- H J Lu's document *ELF: From The Programmer's Perspective* `<ftp://tsx-11.mit.edu/pub/linux/packages/GCC/elf.latex.tar.gz>` contains much useful and more detailed information on programming with ELF. If you aren't LaTeX-capable, it is also available as PostScript.
- There is a manual page for `dlopen(3)` supplied with the `ld.so` package.

6 Legalese

All trademarks used in this document are acknowledged as being owned by their respective owners. *(Spot the teeth-gritting irony there...)*

The right of Daniel Barlow to be identified as the author of this work has been asserted in accordance with sections 77 and 78 of the Copyright Designs and Patents Act 1988. *(Proof by assertion ... it feels just like Usenet)*

This document is copyright (C) 1995 Daniel Barlow <`daniel.barlow@sjc.ox.ac.uk`> It may be reproduced and distributed in whole or in part, in any medium physical or electronic, as long as this copyright notice is retained on all copies. Commercial redistribution is allowed and encouraged; however, the author would like to be notified of any such distributions.

All translations, derivative works, or aggregate works incorporating any Linux HOWTO documents must be covered under this copyright notice. That is, you may not produce a derivative work from a HOWTO and impose additional restrictions on its distribution. Exceptions to these rules may be granted under certain conditions; please contact the Linux HOWTO coordinator at the address given below.

In short, we wish to promote dissemination of this information through as many channels as possible. However, we do wish to retain copyright on the HOWTO documents, and would like to be notified of any plans to redistribute the HOWTOs.

If you have questions, please contact Greg Hankins, the Linux HOWTO coordinator, at `gregh@sunsite.unc.edu` via email, or at +1 404 853 9989.