The Linux ELF HOWTO

Daniel Barlow <daniel.barlow@linux.org> v1.29, 14 July 1996

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 you should upgrade, (2) How to upgrade to ELF-capability, and (3) what you can do then. After a fairly long fallow period in which I have been pretending to do academic work, it has recently been overhauled to give current information for Linux 2.0.

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 previously used, 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-2.17.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 Java, among other things (it's a kicker for many kinds of interpreters). 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. Note that the speed difference only applies to code that is part of shared libraries. For applications or kernels there is no speed difference between a.out and ELF.

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 may be possible to run an application for another x86 Unix under Linux (it depends on the application), 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 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 (or has already). Current releases of the C library and GCC are compiled only for ELF, and other developers are moving ELFwards too.

Many people are concerned about stability (justifiably so, even if it's not so much fun). ELF on Linux has existed since August 1994 and has been publically available since May or June 1995; the teething troubles are probably out of the way by now. You should allow for the possibility of breaking things — as you would with any major upgrade — but the technology that you're upgrading to is no longer bleeding edge.

For a system on which any development is done, or on which you want to run other people's precompiled binaries, ELF is pretty much a necessity these days. Plan to switch to it when you upgrade to version 2.0 of the kernel.

1.3 How to convert to ELF

When this HOWTO was first written, there was only one way, and it was the way described here. These days there are high-quality upgradable distributions available — unless you have invested significant time in setting up your machine exactly how you like it, you might find that a backup of all your own data and a reinstall from a recent Red Hat or Debian release is more convenient than messing about with the assorted libraries and compilers described here.

I must stress this. 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), but there are a multitude of errors that you can make which will probably leave you with an unbootable system. If you are not comfortable with upgrading shared libraries, if the commands `ldconfig` and `ldd` mean nothing to you, or if you're unhappy about building packages from source code, you should consider the 'easy option'. Even if this description isn't you, think about it anyway — if you want a 'fully ELF' system, *somebody* is going to have to recompile all the binaries on it.

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.

This intelligence is centralised in a *dynamic loader*, which exists in only one — or two — places on the system. For a.out programs, it's called `/lib/ld.so`, and for ELF programs it's `/lib/ld-linux.so.1`. 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 the appropriate dynamic loader in, and leave that 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` (`ld-linux.so.1`; whatever) 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` or `ld-linux.so.1` may cause *every dynamically linked program on the system to stop working*. This is generally regarded as a Bad Thing.

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:

- 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, any libraries that you move from `/lib` must end up somewhere else on the root disk, not on `/usr`. I used `/lib-aout` in the instructions that follow.

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**. 1.2.13 works. 2.0.0 (the most recent at the time of writing) also works, as do most of the 1.3 series, though the point of running old 'experimental' kernels is anyway questionable now that 2.0 is here.

- 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 were following the early ELF development, or you installed certain versions of Slackware (none of the current ones, admittedly) 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/`. It is especially important to check this when you install new programs; if you have `/etc` nearer the front of the search path than `/sbin` you'll get odd failures due to running the old versions when you weren't expecting to.

- It's a good idea to pick a time when nobody else is using the computer, or to take it single-user. It might be a good idea to reboot it off a floppy so that a mistake doesn't leave you stuck, but personally I like to leave a *small* element of fun ...

2.3 Ingredients

Anything in the following list that I describe as being "on `tsx-11`" can be found in `<ftp://tsx-11.mit.edu/pub/linux/packages/GCC/>`, `<ftp://sunsite.unc.edu/pub/Linux/GCC/>`, and at many mirrors. 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.

Even if you habitually compile things from source, I'd advise you to go for the binary versions where I've indicated, unless you *really* have no use for your hair. Most of them are not set up for 'crosscompiling' on an a.out-based system, and you are probably lining yourself up for major grief if you try.

2.3.1 Absolute essentials

- `ld.so-1.7.14.tar.gz` — the new dynamic linker. Contains both source and binaries. Note that forthcoming versions of this will require kernel ELF support even for a.out binaries; if you get 1.8.1 or later instead of the version listed, make sure that the kernel you're running was compiled with ELF support *before* you install this.

- `libc-5.3.12.bin.tar.gz` — the ELF shared images for the C and maths libraries, plus the corresponding static libraries and the include files needed to compile programs with them. Source is also available if you like it, but it takes ages to compile, and probably won't at all unless you already have an ELF system.

- `gcc-2.7.2.bin.tar.gz` — the ELF C compiler package, which also includes an a.out C compiler which understands the new directory layout. If you want to build gcc yourself (which you'll probably find is simpler when you're already running ELF), you're recommended to apply `gcc-2.7.2-linux.diff.gz` to the GNU sources first.

- `binutils-2.6.0.12.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. Note that the vanilla GNU binutils (e.g. from `prep.ai.mit.edu`) are not an acceptable substitute; if you really want to compile this yourself you'll need to use the patched-for-Linux `binutils-2.6.0.12.tar.gz` package instead of the GNU one.

- `ncurses-1.9.9e.tar.gz` — this is an SVR4-compatible curses library, which is henceforward deemed to be the 'standard curses library' for Linux. The source is available from GNU sites such as `<ftp://prep.ai.mit.edu/gnu/>` and also from `<ftp://ftp.netcom.com/pub/zm/zmbenhal>`, and there is a binary package on `tsx-11`. By the time you get to install this you will have a fully functional ELF development system, so I recommend the source package if you have any kind of compilation horsepower. That may just be me, though.
- `gdbm-1.7.3.tar.gz` is a set of database routines that use extensible hashing and work similarly to the standard UNIX dbm and ndbm routines. The source is available from GNU sites such as `<ftp://prep.ai.mit.edu/gnu/>`; you also need a patch `<ftp://ftp.uk.linux.org/pub/Linux/libc/non-core/gdbm.patch>` to make shared libraries out of it. That patch also fixes a couple of other things (a one-character typo in the Makefile and a predisposition to use the wrong kind of file locking).

2.3.2 Others

These are other libraries and files which aren't strictly essential, but that you might want to get anyway. This list contains only packages that need to be upgraded to work in an ELF-useful fashion. Later in this document is another list of programs which will continue to work but which you'll have to tweak/upgrade if you want to recompile them in ELF. If your net access involves high-latency links (like, say, a five-minute walk with a box of floppy disks), skip forwards and check that one too before you set out :-)

- The **a.out compatibility** library package, `libc.so-4.7.6`. This is listed as 'optional' because your existing a.out libraries of whatever vintage will continue to work fine with your existing binaries. You might find that you need this if you plan to continue developing in a.out for whatever reason.
- **BSD curses**. If you find binaries which require `libcurses.so.1`, this is the old BSD curses library. They're probably quite rare, which is fortunate as I can't presently find a (source code) copy of the library. It's probably best to recompile programs like this to use ncurses; if this is not an option, there is a binary `libcurses.so` in the `libc-5.0.9.bin.tar.gz` on `tsx-11` mirrors.
- **Berkeley db**: the new 4.4BSD `libdb` database routines. The source can be had from `<ftp://ftp.cs.berkeley.edu/ucb/4bsd/db.1.85.tar.gz/>`, and the patch for Linux shared libraries is `<ftp://ftp.uk.linux.org/pub/Linux/libc/non-core/db.patch>`
- **C++ stuff.** The `gcc` package comes with `g++`, but you'll also need `libg++-2.7.1.4.bin.tar.gz` to compile any useful C++ software. I don't use C++ myself, but I understand that it is nontrivial to build this from source, hence the binary recommendation.
- **GNU-compatible termcap**. The conversion to ncurses din't happen simultaneously with the move to ELF — you might find that you want to run other people's programs that were built using this library, and for some applications you might wish to continue using it. `gdb` is a legitimate example. If you intend to debug shared libraries and you think that gdb is getting confused about the ones that it's linked with itself, you probably want a statically linked copy of it; in this case, you'll find that a real termcap is a lot smaller than the termcap-compatible routines in ncurses.

  `termcap-2.0.8.tar.gz` is available from `tsx-11`. This is *not* GNU Termcap, but it is completely compatible (the differences are in the error checking, apparently). This is a source code package.

- **MAKEDEV**. In some incarnations, this utility removes existing entries for devices before recreating them. This is Bad News if it removes `/dev/zero`, which causes some versions of `ld-linux.so.1` to break. Find a new version at
`<ftp://sunsite.unc.edu/pub/Linux/system/Admin/MAKEDEV-C-1.5.tar.gz>` or
`<ftp://sunsite.unc.edu/pub/Linux/system/Admin/MAKEDEV-2.2.tar.gz>`.

- `modules-2.0.0`. If you use modules, the upgrade to binutils which you're shortly about to perform will break all versions of the modules utilities older than 1.3.69. New modules utilities can be had from `<http://www.pi.se/blox/>`.

- The **X window system** includes a lot of shared libraries. As your new programs will be ELF, and ELF programs cannot use a.out libraries, you'll need a new X installation if you want to do any X development. 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. Note that XFree86 binaries currently require an ELF shared termcap library (`libtermcap.so.2`) to be installed.

  If you use Motif, you may also need to contact your vendor, to investigate whether they will supply ELF Motif libraries. I don't use it; I can't help here.

- If you're upgrading to Linux 2.0 at the same time as going ELF, don't forget also to check the `Documentation/Changes` file that comes in the kernel source, to find out what else you'll need.

2.4 Rearranging your filesystem

Sooo... Note that in all that follows, when I say 'remove' I naturally mean 'backup then remove' :-). Take a deep breath ...

1. **The essentials — binary installation**

2. 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
   ```

3. Untar the dynamic linker package `ld.so-1.7.14` in the directory you usually put source code, then read through the `ld.so-1.7.14/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 that's scant consolation when you're looking for the rescue disk)

4. 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.

5. Move all the a.out libraries in `/usr/lib` and `/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.

6. 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 there are any others): 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`

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

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

9. You need to clean up your `/usr/include` hierarchy. On an average system, some of the files in here are 'core' functionality and come with libc, while others are from other packages that you or your distribution builder have installed. Given this mess, I suggest you remake it from scratch; rename it to `/usr/include.old`, then unpack `libc-5.2.18.bin.tar.gz` by untarring it from the root directory.

10. Install the binutils package. `tar -xvzf binutils-2.6.0.12.bin.tar.gz -C /` is one perfectly good way to do this.

11. The gcc package expects to be untarred from root. It installs some files in `/usr/bin` and lots more in `/usr/lib/gcc-lib/i486-linux/2.7.2` and `/usr/lib/gcc-lib/i486-linuxaout/2.7.2`. Use

    ```
    $ tar ztf gcc-2.7.2.bin.tar.gz
    ```

    to see what's in it, backup anything that it overwrites that you feel you may want to keep (for example, if you have Gnu ADA installed you will probably want to keep `/usr/bin/gcc`), then just do

    ```
    # tar -zxf gcc-2.7.2.bin.tar.gz -C /
    ```

    At this point, you should be able to run `gcc -v` and compile test programs. Try

    ```
    $ gcc -v
    Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2/specs
    gcc version 2.7.2
    $ gcc -v -b i486-linuxaout
    ```

```
Reading specs from /usr/lib/gcc-lib/i486-linuxaout/2.7.2/specs
gcc version 2.7.2
$ ld -V
ld version 2.6 (with BFD 2.6.0.2)
  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 correctly.

Finished? Not quite. You still have all the ‘non-core’ libraries to install, and a fair amount of mucking about with symlinks. Onwards...

Symlinks

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 probably 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/2.7.2/cpp .
```

13. When you moved `/usr/include` to `/usr/include.old`, you lost the symlinks into the kernel sources. Run

```
# cd /usr/include
# ln -s ../src/linux/include/linux .
# ln -s ../src/linux/include/asm .
```

(assuming you have kernel source in `/usr/src/linux`; if not, season to taste)

14. 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.

Rejoice!

By this time you should have a (more or less) fully functioning ELF development system. Stand back and celebrate quietly for a few minutes.

Essential source code packages

15. **ncurses installation** is a fairly long job, though most of of the time can be spent reading Usenet while it builds. After unpacking the tar file, read the `INSTALL` file pretending that you are 'a Linux ... distribution integrator or packager'; that is, you probably want to be configuring it with a command like

    ```
    $ ./configure --with-normal --with-shared --disable-termcap --enable-overwrite --prefix=/usr
    ```

    Take heed also of the comments about the default terminal type; in 1.3 and 2.0 kernels this is set to `linux` at boot time, but you may find that you need to edit `/etc/inittab` to avoid having it set back to `console` by `getty`.

    If you do not have `/usr/lib/terminfo` on the root disk you will have to fiddle with the 'fallback' support in ncurses. This is documented in the `INSTALL` file mentioned above, and is simple but tedious (due to the necessity of building the library twice). If you're happy with having `linux` and `vt100` as fallbacks, there is a ready-prepared `fallback.c` at `<ftp://ftp.uk.linux.org/pub/Linux/libc/non-core/fallback.c>` which you can copy over the existing one.

    After you have installed ncurses, you'll have to get messy in `/usr/lib` — it does some non-optimal things that are simplest to clear up by hand. Note the weird discrepancy between the version numbers; this is ugly but not actually detrimental to human health.

    (a) `/usr/lib/libncurses.so.1.9.9e` should be moved to `/lib` so that curses programs which run in single-user mode will continue to do so. If you have `/usr/lib` on the root partition, this is unnecessary but will do no harm.

    (b) In `/lib`, make a link to `libncurses.so.1.9.9e` called `libncurses.so.3.0`.

    (c) You also need links `/usr/lib/libncurses.so`, `/usr/lib/libcurses.so` and `/usr/lib/libtermcap.so` which should all point to `/lib/libncurses.so.3.0`.

    In brief for the hard of thinking, that little lot was

    ```
    # cd /lib
    # mv /usr/lib/libncurses.so.1.9.9e .
    # ln -s libncurses.so.1.9.9e libncurses.so.3.0
    # cd /usr/lib
    # ln -s /lib/libncurses.so.3.0 libncurses.so
    # ln -s /lib/libncurses.so.3.0 libcurses.so
    # ln -s /lib/libncurses.so.3.0 libtermcap.so
    ```

16. **gdbm** installation. Unpack the source code in a source code directory, apply `gdbm.patch`, and look over the `README` and `INSTALL` files.

    The build process should go something like:

```
$ tar zxf gdbm-1.7.3.tar.gz
$ patch -p0 < gdbm.patch
$ cd gdbm-1.7.3
$ ./configure --prefix=/usr
$ make
$ make progs
$ su
# make install
# make install-compat
# cd /usr/lib
# ln -s libgdbm.so.1 libgdbm.so
# ln -s libgdbm.so.1 libgdbm.so.2
# ldconfig
```

The last step is for backward-compatibility; some current distributions use `libgdbm.so.2` which is exactly the same code as `libgdbm.so.1`, but misnumbered for historical reasons.

Optional source code packages. In general, you can just install these according to their instructions, so I won't repeat them. There are two exceptions, though:

17. If you want the **GNU-ish termcap** (strictly speaking, optional; in practice, necessary to use XFree86 binaries) it also needs to be built from source, but shouldn't require anything more complex than

```
$ tar zxf termcap-2.0.8.tar.gz
$ cd termcap-2.0.8
$ make
$ su
# cp libtermcap.so.2.0.8 /usr/lib
# ldconfig
```

I recommend that you *don't* `make install`, as this would overwrite bits of the ncurses installation. If you need to actually compile things against this library, as opposed to just running binaries that were made with it, think about putting the header files and static libraries somewhere nonstandard, and using `-I` and `-L` flags when you compile the said things. The vagueness of this description should make it plain that continued use of termcap is 'discouraged' unless you have a good reason.

18. For `libdb`, it goes something like:

```
$ tar zxf db.1.85.tar.gz
$ patch -p0 <db.patch
$ cd db.1.85/PORT/linux
$ make
$ su
# mkdir /usr/include/db
# ldconfig
# cp libdb.so.1.85.3 /usr/lib ; ( cd /usr/lib && ln -s libdb.so.1 libdb.so )
# cp ../../include/*.h /usr/include/db
```

Note that

- you're not applying `PORT/linux/OTHER_PATCHES`, because it's subsumed by this patch
- you're installing the header files somewhere other than `/usr/include` — they conflict with the ones that gdbm uses. To compile programs that want `libdb` you must add `-I/usr/include/db` to the C compiler's command line.

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`,`libncurses.so.1`,`libtermcap.so.2`, (ELF). Again, these are symlinks. Check the files that they point to.

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.
- More symlinks. For each library in `/lib` or `/usr/lib`, there should be a symlink in here. The link's name should be the real filename, minus the version number. For example, for `libc`,

  ```
  lrwxrwxrwx   1 root     root           14 May  2 20:09 /lib/libc.so.5 -> libc.so.5.3.12
  -rwxr-xr-x   1 bin      bin        583795 Apr 25 06:15 /lib/libc.so.5.3.12
  lrwxrwxrwx   1 root     root           12 Oct 27  1995 /usr/lib/libc.so -> /lib/libc.so.5
  ```

  These links are used by `ld` at link time.
- `libbsd.a`, `libgmon.a`, `libmcheck.a`, `libmcheck.a` and one `lib*.a` file for every ELF shared library in `/lib` and `/usr/lib`. ELF static libraries. The ones that duplicate 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. You *will* need to keep them if you actually want to debug the libraries themselves.
- `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 do this.
- `ldscripts` is a symbolic link to `/usr/lib/ldscripts`

2.5.8 /usr/lib/gcc-lib/i486-linux/2.7.2

- This directory contains a version of gcc 2.7.2 set up to compile ELF programs.

2.5.9 /usr/lib/gcc-lib/i486-linuxaout/2.7.2

- This directory contains a version of gcc 2.7.2 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. Note that you need to keep it if you want to build unpatched 1.2 series kernels.

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.

`bad address`

on attempting to run anything ELF. You're using kernel 1.3.x, where x<3. Don't. They're probably the buggiest Linux kernels on the planet anyway. Upgrade to 2.0 or downgrade to 1.2.13. Some people also report kernel panics in similar circumstances; I haven't investigated, chiefly as I can think of no reason for wanting or needing to run development kernels and not keeping up with the releases.

`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.

`make: *** No targets specified and no makefile found. Stop.`

indicates that you haven't patched and recompiled `make`, or that you still have an old version of it elsewhere on the system.

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

(or any other file that you try to run) 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.

`_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.

3 Building programs

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 Building in a.out

You may have a need to continue to build programs in the old a.out format. For 'normal' programs all you need to do to use the a.out compiler is specify the flag `-b i486-linuxaout` when you call gcc, and `-m` `i386linux` when (if) you call ld. If you need to build a.out DLL shared libraries still, you have my sympathy. To the best of my knowledge, the short answer is that it doesn't work. Please mail me if you know different.

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 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 (i.e. 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**. Nowadays, 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/>
- **e2fsutils**. The Utilities for the Second Extended File System versions 0.5c and later compile unchanged in ELF.
- **Emacs**. There are two potential problems here. (i) 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. (ii) If you build some versions of emacs against ncurses, it will fail unless you first edit `src/s/linux.h` in the emacs distribution to add the line `#define TERMINFO` somewhere near the top. This is not necessary for 19.31, but is for XEmacs 19.13. Apparently it will be fixed in 19.14.

- **gdb 4.16**. Your existing copy of gdb will continue to work just as well as it always has done in the past, but the shared library support in 4.16 is a lot better, so if you want to debug programs that do weird things in that area, this is a good upgrade.
- **The Kernel**. Kernel versions 2.0 and greater work fine with ELF; you have to say 'yes' to both of

```
Kernel support for ELF binaries (CONFIG_BINFMT_ELF) [Y/m/n/?]
Compile kernel as ELF - if your GCC is ELF-GCC (CONFIG_KERNEL_ELF) [Y/n/?]
```

  when you run `make config` (this is also the case for most of the 1.3 series). If you are using 1.2 still, see the 'patch' list below.
- **perl 5**. Perl 5.001m and later will compile unchanged on an ELF system, complete with dynamic loading. Current versions of Perl are available from CPAN (Comprehensive Perl Archive Network) sites: see `<ftp://ftp.funet.fi/pub/mirrors/perl/CPAN>` for the closest one to you.
- **ps** and **top**. Procps 0.98 and greater will work with ELF (earlier versions also work, but can't read the kernel to find WCHAN names, if you care about them). Note that 2.0 series kernels require procps 0.99a or greater anyway.
- The `cal` program in **util-linux 2.2** doesn't work. Upgrade to *version 2.5* `<ftp://tsx-11.mit.edu/pub/linux/packages/utils>` or later.
- **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

- **file**. This works anyway, but can be improved: `<ftp://ftp.uk.linux.org/pub/Linux/libc/non-core/file.patch>` adds support for identifying stripped and mixed-endian ELF binaries.
- `make-3.74` — either get the source code from a GNU site and apply the patch that comes with the libc-5.3.12 release notes, or get the binary `make-3.74.gz` from `tsx-11`. There is a bug in GNU make which only manifests with new ELF libc versions — it's actually a dependency on a bug in old versions of the GNU libc, which was also present in Linux libc until recently. If you keep your old a.out `make` program it will continue to work, but if you want an ELF one you need the patch.

  The GNU Make developers know about the bug, and one day will release a fixed version.
- **The 1.2.x Kernel**. You have three options:

  1. Patch the Makefile slightly to use the a.out compiler. `cd /usr/src/linux/`, cut the following patch out, and feed it into `patch -p1`. Or just edit the Makefile manually using this as a guide; it's clear enough (delete the lines marked with a `-` and add the ones with a `+`.

```
diff -u linux-1.2.13/Makefile.orig linux/Makefile
--- linux-1.2.13/Makefile.orig  Wed Aug 16 20:53:26 1995
+++ linux/Makefile      Fri Dec  8 16:19:49 1995
@@ -12,9 +12,9 @@
 TOPDIR := $(shell if [ "$$PWD" != "" ]; then echo $$PWD; else pwd; fi)

-AS     =as
-LD     =ld
-HOSTCC =gcc -I$(TOPDIR)/include
-CC     =gcc -D__KERNEL__ -I$(TOPDIR)/include
+AS     =/usr/i486-linuxaout/bin/as
+LD     =ld  -m i386linux
+HOSTCC =gcc -b i486-linuxaout -I$(TOPDIR)/include
+CC     =gcc -b i486-linuxaout -D__KERNEL__ -I$(TOPDIR)/include
 MAKE   =make
 CPP    =$(CC) -E
 AR     =ar
```

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). This can be had from `<ftp://ftp.cdrom.com/pub/linux/slackware_source/kernel-source/v1.2/linuxelf-1.2.13.diff.gz>`.

   If you are using an ELF distribution (RedHat 2.1, Slackware 3) which comes with a 1.2 series kernel, you will probably find that this patch or one similar has been applied already.

   The best idea, hoever, is probably

3. Upgrade to 2.0! 1.2 was never really intended for ELF anyway.

You will have other problems compiling 1.2.13 with gcc 2.7.2 and above; there was a bug in `asm/io.h` which is only detected by gcc 2.7.2. You will need the patch `<ftp://ftp.uk.linux.org/pub/Linux/libc/misc/io.h>`.

5 Further information

- The *GCC-HOWTO* `<GCC-HOWTO.html>` contains much useful information about development on Linux (at least, I think it does; I maintain it). It should be available from the same place as you found this, which is why the link above is relative.

- The `linux-gcc` mailing list (which is also the `linux.dev.gcc` newsgroup, if you have a `linux.*` news feed) 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://www.linux.ncm.com/linux-gcc/>`.

- There's a certain amount of information about what the linux-gcc list is doing at my *linux-gcc web page* `<http://ftp.uk.linux.org/ barlow/linux/gcc-list.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>`

- 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.
- Information about the **ncurses** library and the terminfo database is available from *Eric Raymond's ncurses resource page* `<http://www.ccil.org/ esr/ncurses.html>`.
- There is a manual page covering `dlopen(3)` and related functions, which is supplied with the `ld.so` package.

6 Miscellanities

6.1 Feedback

is welcomed. Mail me at daniel.barlow@linux.org. My PGP public key (ID 5F263625) is available from my *web pages* `<http://ftp.uk.linux.org/ barlow/>`, if you feel the need to be secretive about things.

If you have a question that you feel this document should have answered and doesn't, mail me. If you have a question which probably shouldn't be answered here but you think I might know the answer anyway, you might want to try posting to an appropriate `comp.os.linux.*` newsgroup first; I usually answer mail eventually, but I have been known to lose it on occasion.

Anyone found adding my name to junk email lists will pay dearly for it.

6.2 Translations

If you wish to translate this document, please go right ahead, but do tell me about it! The chances are (sadly) several hundred to one against that I speak the language you wish to translate to, but that aside I am happy to help in whatever way I can.

Translations that I know of are:

- *Italian* `<http://www.psico.unipd.it/ildp/docs/HOWTO/ELF-HOWTO.html>`, by Favro Renata. (Other HOWTOs are also available in Italian from `<http://www.psico.unipd.it/ildp/docs/HOWTO/INDEX.html>`.
- Kojima Mitsuhiro has produced a Japanese translation, available from `<http://jf.gee.kyoto-u.ac.jp/JF/index.html>`.

6.3 Legal bits

All trademarks used in this document are acknowledged as being owned by their respective owners. Yow!

If you have questions, please contact Greg Hankins, the Linux HOWTO coordinator, at `gregh@sunsite.unc.edu`.