3.3 Release Notes:
Multiple Architecture Binaries (fat files)

This file contains release notes for the 3.1 and later releases of NEXTSTEP relating to building executable files for multiple architectures. There are no additions for Release 3.3 or 3.2.

The Basics:

The concept of fat files is to allow the same file to contain the executable for more than one architecture. When the file is executed the kernel will pick the one that matches the machine.

The rule of thumb here is that the default action when dealing with a fat file or doing an operation that results in an architecture dependent output is to is to do the action or produce the output for the machine the action was taken on. Thus when a fat file is executed on a machine the part for the architecture that matches the machine is

used. When compiling a source file, the default target architecture is the same as the architecture of the machine that performs the compilation.

Cross compiling:

The target architecture to be compiled for is specified by the flag ª-arch nameº, where name is the name of the target architecture (see arch(3) for supported architectures). More than one ª-arch nameº flag may be used with the compiler driver cc(1) to build a fat object or executable file.

The compiler tools such as nm(1) and size(1) also accept the ª-arch nameº flags. For the most part these tools also support the flag ª-arch allº when dealing with fat files (however, the compiler driver cc(1) doesn't support this flag). ld(1) itself does not take multiple arch flags. So to run an ªld -rº on a set of objects files for multiple architectures cc(1) can be used as follow:

```
cc -nostdlib -r -arch m68k -arch i386 ... a.o b.o ...
```

The format:

The layout of a fat file is a simple wrapper design, which has a simple header with the contents of each component appearing continuously in the fat file. The contents of each component in the fat file is always byte-for-byte the same as it would be if it were not in a fat file.

The two header structures are defined in <mach-o/fat.h>, and the structures always

appear in big endian byte order.

The Warts:

There are a number of warts in the integration of building fat files with make(1) and other standard UNIX tools in some areas. However, performing a ªmake cleanº followed by a ªmake allº should always work. The basic problem is that make(1) does not know about fat files and if different invocations of make(1) are run with differing ª-arch nameº flags make will not do the ªrightº thing by itself. Project Builder and the supporting application Makefiles provide a simple user interface for getting the right behavior when building NeXT applications, but make itself doesn't support the general case.

Archive libraries

The integration of fat archive libraries with UNIX tools like ar(1) and make(1) has its limitations. The problem is how to define a proper fat archive library. A proper fat archive library is multiple archives containing thin object files not a single archive containing fat object files. When building fat archive library with cc(1) and ar(1) then running ranlib(1) on the result causes the correct form of a fat archive library to be built, by following some well defined steps. First as each source is compiled with multiple -arch flags to the compiler, fat object files are built. Then when the archive is created or added to with ar(1), an archive with fat object files is created. Finally when ranlib(1) is run on the archive to make it into a library, the fat object files are broken out and multiple archives are created in one fat file with multiple tables of contents one for each archive.

Problems come about when attempting to modify a fat archive library with the UNIX tool ar(1), as ar(1) does not know about archives in fat files. ar(1) was not taught about archives in fat files as it would be difficult to define the functionality with respect to fat or thin input files that did not match the existing archive. So NeXT recommends the use of a new tool libtool(1) that correctly builds libraries (fat or thin as needed) from a list of any types of files containing objects.

This problems can also show up with make(1), which does not know about fat archive libraries. An example of such a use would be when make tries to process a rule like:

```
libx.a(x.o): x.c
```

and libx.a is not an archive file but a fat file containing archives.