HP-UX

Large Files

White Paper

Version 1.4

1. Introduction

1.1 Abstract

With the release of HP-UX 10.20, Hewlett-Packard introduces support for files over 2 GB's in size on 32-bit machines. In past releases the maximum size for a file has been 2 GB's. This implementation adheres to the specifications outlined by the Large File Summit, which is a group of OS vendors and application developers. Large files are supported in many areas of the HP-UX operating system, including standard library calls, the kernel filesystem interface, and many appropriate commands.

HP-UX 11.0 is the first HP-UX release to be delivered as a 64-bit OS. The information in this document was originally designed for using 64-bit values on a 32-bit OS. The items discussed in this paper still apply to the 32-bit OS version of HP-UX 11.0, and the 32-bit applications (including the ones supporting large files) on a 64-bit OS. Many of the items will also apply to 64-bit applications on a 64-bit OS, however, where differences occur, these will be noted.

1.2 Purpose of Document

The purpose of this document is to communicate the availability of large files, the implications of enabling large files on your system, and to give developers guidance on how to account for and take advantage of large files.

1.3 Intended Audience

This document is intended for system administrators of HP-UX systems that need to provide large files on their systems, as well as developers of applications and libraries that will either take advantage of large files or will need to co-exist with large files.

1.4 Overview of Document

"Section 3. Enabling Large Files" and "Section 5. Commands Support of Large-Files" will be of particular interest for system administrators as they describe how to enable large files on your system and what commands will work with large files. "Section 6. Application Development" is required reading for any developer that will have an application or library on a large file enabled system.

1.5 Related Documentation

The following list of references was used in the preparation of this white paper. The reader is urged to consult them for more information. All of these documents are available via http at:

http://www.sas.com/standards/large.file/index.html

- [1] "Overview of the Large File Support effort"
- [2] "Draft Specifications for Large File Support"
- [3] "HP's Large Files and the Large File Summit December, 1995" - (See Appendix 1)

2. Overview

This chapter gives a list of product features and covers several key points for application developers. This chapter does not describe impacts to applications, these are covered later in the document.

2.1 Product Features

Included below is a list of the features provided by large files for the HP-UX:

- Support file sizes greater than 2 GB on HP-UX.
- Remain binary compatible. No change required to existing applications as long as they do not access large files.
- Provide new data types, data structures, and macros to support large files.
- In addition to existing POSIX API, support new API for HP customers.
- Support API error handling: an error is returned whenever an API cannot return the correct result of an operation.
- Provides a new compile option, `_FILE_OFFSET_BITS`, that can be used to set the compile environment to 64-bits or 32-bits. (See Section 6.1.2 Compile Environments)
- Provides two new interfaces, `ftello` and `fseeko`, that are made available by the `_LARGEFILE_SOURCE` compile option. (See Section 6.1.1.3 New Non-POSIX API)
- Provide one additional compile option `_LARGEFILE64_SOURCE` to provide the new API. (See Section 6.1.1.4 New 64-bit Specific Interfaces and Data types)
- Provides two announcement macros, `_LFS_LARGEFILE` and `_LFS_LARGEFILE64`, to determine if `_LARGEFILE_SOURCE` and `_LARGEFILE64_SOURCE` are supported. (See Section 6.1.4 Determining if large-files are supported)
- Supported by appropriate HP-UX commands.
- Support POSIX compliance.

2.2 Product Limitation

The HP-UX 10.20 release supports a maximum file size of 128GB's. It is expected that the maximum file size will increase in future releases of HP-UX. Large files are not supported by the NFS or DFS filesystems in HP-UX 10.20. Large File support has been added for NFS in HP-UX 11.0.

2.3 Application Interfaces to Large Files

Some steps have to be done before an application can access a large file.

- The application source code needs to be modified properly.
- The application needs to be compiled with different compile options to access large files.

For applications that wish to support large files, two interface mechanisms are provided. First, a set of new system calls (new API) are available for applications. Second, a 64-bit compile environment (`_FILE_OFFSET_BITS=64`) is provided for application development, which modifies the size of various data types and structures in the existing POSIX API to support large files.

New 64-bit API

HP-UX adds a set of 64-bit data types, system calls, and libc routines in the form of *interface*64() to support large files. This new API is like the existing POSIX API, except that all applicable file sizes, offsets, and block counts are 64 bits. It also contains a new version of `open()` (`open64()`) that succeeds on files of any size. The new API was defined by a group of OS and application suppliers.

An application using the new API will be compiled in the 32-bit compile environment. This means compiler views data types such as `off_t` as 32-bit and it is left to the application to pass around 64-bit data types within the program properly to access large files.

In order to avoid name space pollution in user space, the `_LARGEFILE64_SOURCE` compile option must be specified to use the new API. Please note that the `_LARGEFILE64_SOURCE` compile option will include the interfaces provided by the `_LARGEFILE_SOURCE` compile option.

The new API is intended for C programmers; if you use another language, please refer to the man pages and/or compiler documentation.

64-bit compile environment

Note: 64-bit compile environment refers to compiling a 32-bit application with 64-bit extensions; it does not refer to a 64-bit application that will only run on a 64-bit OS.

An application can be compiled in a 64-bit compile environment by setting the `_FILE_OFFSET_BITS` compile option to 64 (`_FILE_OFFSET_BITS=64`). This mechanism automatically changes the size of various data types and structures from 32 bits to 64 bits. For example, `off_t`, normally 32 bits, is 64 bits in this environment, and `lseek` in this environment expects a 64-bit offset of type `off_t`.

For HP-UX 11.0,`_FILE_OFFSET_BITS` is set to 64 by default when compiling a 64-bit application (using `+DDLP64`).

Complete details of the application conversion guideline can be found in “Section 6. Application Development.”

New non-POSIX API

HP-UX adds two new libc routines, `fseeko()` and `ftello()`, as a part of the large files implementation because `ftell()` and `fseek()` can not be extended to support large files. These interfaces are provided in the 32-bit and 64-bit compile environment by invoking the `_LARGEFILE_SOURCE` compile option. Please note that the `_LARGEFILE64_SOURCE` compile option includes the interfaces provided by the `_LARGEFILE_SOURCE` compile option. More detailed information about these interfaces is available in “Section 6. Application Development“.

2.4 Protecting existing applications through filesystem administration

One feature of large files is to provide safe areas for existing applications that may not correctly behave if they encounter a large file. Various commands allow the system administrator to control whether or not a large file may be created on a file system.

File systems may be marked and mounted, specifying whether or not they are to contain large files. The `fsadm` command allows the system administrator to selectively enable and disable the creation of large files on individual file systems. `mkfs` may also be used to enable large files when creating a file system.

Once a file system has been marked to enable large files, using the above commands, `mount` is then used to mount the file system. All of these commands accept the `largefiles` and `nolargefiles` options.

Any attempt to create a large file on a file system that has not been appropriately enabled with either `fsadm` or `mkfs` will result in the same error as previous releases of HP-UX (prior to 10.20). This option protects existing applications and allows them to continue working without any source changes.

Complete details of the administration commands implementation may be found in “Section 3. Enabling Large Files“.

3. Enabling Large Files

3.1 Default on system is small files

Large files is a technology that must be explicitly enabled. A system will not support large files just because it has been updated to a release of HP-UX that supports large files. The advantage of this is that if large files are not needed, then they do not need to be enabled on the system and everything will continue to work as it has in the past.

3.2 Creating a large-files filesystem

Creating a large files filesystem can be done with the `mkfs` command or the `newfs` command. The `newfs` command is a friendly interface to the `mkfs` command, and they both use the same options to create large-files and no-large-files filesystems. As of this release, the default behavior of these commands is to create a no-large-files filesystem. However, this default could be changed in future release of HP-UX. Therefore, it is a good idea to explicitly set either the large files or no large files option.

The following examples show how to create a large-files filesystem.

- `/usr/sbin/mkfs -F hfs -o largefiles /dev/vg02/rlvol1`
- `/usr/sbin/newfs -F hfs -o largefiles /dev/vg02/rlvol1`
- `/usr/sbin/mkfs -F vxfs -o largefiles /dev/vg02/rlvol1`
- `/usr/sbin/newfs -F vxfs -o largefiles /dev/vg02/rlvol1`

The following examples show how to create a filesystem that will not support large files.

- `/usr/sbin/mkfs -F hfs -o nolargefiles /dev/vg02/rlvol1`
- `/usr/sbin/newfs -F hfs -o nolargefiles /dev/vg02/rlvol1`
- `/usr/sbin/mkfs -F vxfs -o nolargefiles /dev/vg02/rlvol1`
- `/usr/sbin/newfs -F vxfs -o nolargefiles /dev/vg02/rlvol1`

3.3 Changing a file system from one to the other

HP-UX also provides the ability to change a filesystem back and forth between large files and no large files. This is provided by the `fsadm` command which also provides other filesystem administration capabilities. It is important to realize that the conversion of these filesystems must be done on an unmounted filesystem, and `fsck` will be called after a successful conversion.

The following example shows how to convert a no-large-files filesystem to a large-files filesystem.

- `/usr/sbin/fsadm -F hfs -o largefiles /dev/vg02/rlvol1`

While the conversion of a no-large-files filesystem to a large-files filesystem should always succeed, the same is not true for converting a large-files filesystem to a no-large-files filesystem. The latter will only succeed if there are no large files on the filesystem. If even one large file is detected on the filesystem being converted, then the `fsadm` command will not convert the filesystem. Therefore, if it is necessary to convert a large-files filesystem that actually has large files on it to a no-large-files filesystem, the large files must be removed before conversion.

The following example shows how to convert a large-files filesystem to a no-large-files filesystem.

- `/usr/sbin/fsadm -F hfs -o nolargefiles /dev/vg02/rlvol1`

3.4 Mount Protection

The `mount` command has been modified to support large-files filesystems and provides the system administrator with a method of ensuring that no large-files filesystems are mounted on the system. The `mount` command uses the same two options as the `mkfs, newfs`, and `fsadm` commands (`largefiles` and `nolargefiles`). `mount` will not mount a large-files filesystem if the `-o nolargefiles` option is specified. Conversely, the `mount` command will not mount a no-large-files filesystem if the `-o largefiles` option is specified. If no option is provided to `mount`, it will use the state of the filesystem itself to determine if it is mounted as largefiles or nolargefiles. The following table summarizes the expected mount results:

Table 1: Mount Results

Mount Command	Filesystem type	Result
/usr/sbin/mount -F hfs -o largefiles /dev/vg01/lvol2 /tmp/test	no-large-files	Fails
/usr/sbin/mount -F hfs -o nolargefiles /dev/vg01/lvol2 /tmp/test	no-large-files	Pass
/usr/sbin/mount -F hfs -o largefiles /dev/vg01/lvol2 /tmp/test	large-files	Pass
/usr/sbin/mount -F hfs -o nolargefiles /dev/vg01/lvol2 /tmp/test	large-files	Fails
/usr/sbin/mount -F hfs /dev/vg01/lvol2 /tmp/test	no-large-files	Pass: no-large-files
/usr/sbin/mount -F hfs /dev/vg01/lvol2 /tmp/test	large-files	Pass: large-files

4. Backing Up Large Files

Providing the ability to create and modify large files is not enough. HP-UX must provide some basic ability to backup and recover files greater than 2GB. There are a number of tools that perform this task.

4.1 Backup utilities that support large files

The following backup utilities will back up large files:

1. dd
2. fbackup/frecover

Both of these commands require no user intervention to back up large files. If a backup contains large files and an attempt is made to restore the files on a filesystem that does not support large files, the large files will be skipped. `fbackup` and `frecover` will use a new format for this release so a backup tape created on HP-UX 10.20 or later can not be restored on a release of HP-UX prior to 10.20.

4.2 Backup utilities that do not support large files

4.2.1 tar, cpio, pax, ftio

Some of the backup commands, specifically `tar, cpio, pax (tar & cpio` formats), and `ftio` (because it creates `cpio` format archives) are restricted from supporting large files due to standards defined headers in the archives. Although the headers allow archival of files up to 8GB, there is no guarantee that there will be no attempt to restore these files on a system that does not support large files. These commands will therefore support files up to 2GB only. Attempts to archive any files >2GB will fail, and the files will not be added to the archive.

4.2.2 dump, restore

`dump/restore` are large file aware. If `dump` finds the largefile feature bit ‘on’ in the super block, it will issue an appropriate error message and terminate. This will be true even if the file system does not contain any large files. If the largefile (and largeuid) featurebits are not on, `dump` will function as normal, and `restore` will extract from the dump archives.

5. Commands Support of Large-Files

This section of the specification examines the core HP-UX commands support for large files. First, we discuss the rationale and assumptions behind the decisions, then list the commands that will support large files. Finally, we discuss some key implementation details associated with the commands.

5.1 Rationale

For the near future, we expect that large files will be an exception, rather than commonplace. Therefore, it is not necessary to change the entire set of commands to support large files. Rather, we should consider modifying a well thought-out subset of the commands. To help reduce the number of commands that must support large files, we can make the following assumptions about the use of large files:

1. Executables, log files, pattern files and shell scripts will continue to be small. We also do not expect that binary programs will become greater than 2GB any time soon. Log files of this size are usually too large to manage and should be trimmed.
2. Interactive editing will continue to use small files. The operating system may begin to experience obvious performance problems if a user `vi`'s a large file.
3. Printing files will continue to be limited to small files. Although the need has not yet surfaced, printing large graphic files may be required sometime in the future.
4. Mail will continue to support small files only. Mail must be packaged for receipt by any system and there is no way to guarantee a target system's mail handler supports large files. In addition, current protocols do not support multi-megabyte files very well.
5. Tools that package files for exchange between multi-vendor UNIX systems will continue to operate on small files only. Commands such as `shar` cannot guarantee that the target system would be able to unpack the archive. However, these types of tools may need to support large files soon after the feature is available on many other vendors' systems.

To further help us select the commands that support large files, we also consider future standards work. Some of the commands, such as `tar,` are not extensible under current standards to support 128GB files. In such cases, we have chosen not to deviate from the standards. Other commands, such as `pack/unpack,` are marked to be withdrawn from the standards either because they are duplicates of other commands that are considered 'more standard' or no longer have any relevant use. These type of commands have not been enhanced to support large files.

5.2 Commands Supporting Large Files

5.2.1 File System Administration Commands

All of the filesystem administration commands for HFS and JFS support large files. The complete list is as follows:

```
bdf             fsclean         mvdir
chroot          fsdb            ncheck
clri            fstyp           newfs
convertfs       fuser           rmboot
devnm           getext          setext
diskinfo        labelit         sync
disksecn        link            syncer
dumpfs          mkboot          tunefs
extendfs        mkfs            umount
ff              mklost+found    umountall
fsadm           mount           unlink
fscat           mountall        volcopy
fsck                            vxdiskusg
```

5.2.1.1 mkfs

See “Section 3. Enabling Large Files”

5.2.1.2 newfs

See “Section 3. Enabling Large Files“

5.2.1.3 mount

.See “Section 3. Enabling Large Files“

5.2.1.4 fsck

The `fsck(1M)` command repairs damaged large-files filesystem. The external interface of the command is the same, however the brief discussion below describes the changed behavior of the command.

The primary superblock, of the filesystem, is considered to be accurate, after the `fsck(1M)` command verifies it with a secondary superblock. In case the two superblocks do not match, the `fsck(1M)` command does not proceed, with filesystem consistency checks, until the user specifies an alternate primary superblock that is identical to the secondary superblock. This is an existing behavior.

Typically, large files should not manifest in a no-large-files filesystem. However, `fsck(1M)` must recover from this situation. The first scenario uses the interactive mode. `fsck(1M)` finds a large file on a no-large-files filesystem, marks the filesystem dirty and stops. The system administrator then corrects the situation using the `fsadm(1M)` command to turn on the large-files featurebit. The filesystem would then be repaired and available for mounting. This would preserve the large file, if fsck did not find it corrupt in any other way.

In non-interactive mode, the large file on a no-large-files filesystem would be purged. fsck assumes the superblock to be accurate based on its accuracy checks. The probability of a superblock being corrupt is insignificant when compared to the instance of a large file manifesting in a no-large-files filesystem. Consequently, fsck will remove the large file from a filesystem it believes should not contain large files.

5.2.1.5 fsadm

See “Section 3. Enabling Large Files“

5.2.2 Backup Commands

See “Section 4. Backing Up Large Files“

5.2.3 Accounting and Quotas

All of the accounting and quota commands fully support large files through their scanning and reporting mechanisms. They have the ability to recognize large files and report their usages. However, the reporting mechanisms of these commands will continue to create only small files (as stated in the Rationale section of this chapter).

5.2.4 File System User Commands

All of the filesystem user commands support large files. A complete list is as follows:

```
basename        du              pathchk
chgrp           file            prealloc
chmod           find            pwd
chown           getaccess       rm
cp              ln              rmdir
df              ls              touch
dirname         mkdir           whereis
                mv              which
```

5.2.5 Shells

All three of the HP-UX shells, including POSIX shell, Kornshell, and C-shell, support large files through redirection. The default of each shell is to use a large open for redirection. If the ‘open’ occurs on a filesystem that is enabled for large files, a large file may be created. If not, then the file is limited to 2GB. There is no switch (environment variable) that allows the user to toggle between large and small file opens for redirection. This will guarantee that all large file applications will always get the appropriate file descriptor and will not fail because of an environment switch the user may have failed to set. The only area that a user may get into trouble is when a non-largefile application is running in the large file mounted filesystem and encounters a file greater than 2GB through a pipe or redirection. This is considered a very slight risk since most applications of this type do not seek through the data, but rather process it sequentially.

5.2.6 Text Processing Commands

Below is a list of handy commands that may be used to process files that are large. Each will appropriately handle large data files. However, pattern files for commands such as `awk` and `sed` will continue to remain small. There are no new options in this set of commands.

awk	expand	shar
bdiff	fgrep	sort
cmp	fold	split
comm	grep	strings
csplit	head	sum
cut	hyphen	tail
egrep	join	tr
	paste	unexpand
	sed	uniq

5.2.7 Misc. Commands

Below is the list of various commands that do not fall into any particular category, but may be useful with large files. They support large files as well:

cat	page	uudecode
cksum	od	uuencode
compress	tee	wc
mktemp	uncompress	xd
		zcat

5.3 Compatibility

The large file support in the commands does not affect the operation of existing command options. We are not obsoleting or changing any of the command interfaces, only introducing new features and a few new command-line options. There is no impact on binary compatibility.

All of the commands use, in some way, the new interfaces that enable them to operate on large files. This interface exists only on HP-UX systems from HP-UX 10.20 forward. Moving any of these commands that have been enabled for large files backwards to execute on a pre-10.20 system will fail.

5.4 Behavior of commands that do not support Large Files

If commands that do not support large files are run against a large file, the command will return an `[EOVERFLOW]` error and print a message similar to the following:

```
Value too large to be stored in data type
```

6. Application Development

This section discusses key concepts of large-files, conversion choices when dealing with large-files, conversion examples, common trouble areas when converting to large-files, and helpful debugging techniques. This section is important for writing applications to support large files.

6.1 Key Concepts

The size of files for releases prior to HP-UX 10.20 has been limited to 2GB's. This is a direct result of using 32-bit data types for file sizes and offsets. In order to remove this boundary it is necessary to enlarge these types to 64-bit data types. While this can be done for abstract data types, i.e.: `off_t`, it can not be done for integral data types such as a `long` as this will not be changed until HP-UX is moved from a 32-bit OS to a 64-bit OS, which has occurred for HP-UX 11.0. The result of this is that two interfaces, `fseek()` and `ftell()`, are not extensible to large files on a 32-bit OS because of their POSIX definition. In order to address all of these issues and provide large-files on a 32-bit OS, several new compile options have been provided to determine what compile environment to use and what interfaces to make available.

In order to make an existing application large-files aware, it is imperative to understand the two new compile environments, how large-files has been implemented, and the importance of using the correct header files.It is also important for the developer to realize that enabling an application for large files will require some work.

6.1.1 Affected data types, structures, and API's

The following table summarizes the data types and data structures that now have a 32-bit and 64-bit version to support large-files. The structures below have elements that refer to the new data types. Therefore, these structures will be able to support small files in the 32-bit compile environment and to support large files in the 64-bit compile environment. The list of data structures is derived from the list of data types.

6.1.1.1 Data Types And Structures

data types	`blkcnt_t`
	`fsblkcnt_t (unsigned)`
	`fsfilcnt_t (unsigned)`
	`fpos_t (signed)`
	`off_t (signed)`
	`rlim_t (signed)`	

structures	`flock`	`off_t l_start` `off_t l_len`
	`rlimit`	`rlim_t rlim_cur` `rlim_t rlim_max`
	`stat`	`off_t st_size` `blkcnt_t st_blocks`

	`statvfs`	`fsblkcnt_t f_blocks`
		`fsblkcnt_t f_bfree`
		`fsblkcnt_t f_bavail`
		`fsfilcnt_t f_files`
		`fsfilcnt_t f_ffree`
		`fsfilcnt_t f_favail`

	`async_request`	`off_t offset`

6.1.1.2 POSIX API

The following table lists the POSIX API that now have a 32-bit and 64-bit version to support large-files. These API are not broken by definition since they utilize abstract data types to represent file sizes or offsets instead of integral data types.

Table 2: POSIX API

creat()	fgetpos()	fopen()	freopen()	fsetpos()
fstat()	fstatvfs()	fstatvfsdev()	ftruncate()	ftw()
lockf()	lseek()	lstat()	mmap()	nftw()
open()	prealloc()	stat()	statvfs()	statvfsdev()
tmpfile()	truncate()	getrlimit()	setrlimit()	

POSIX API that can not be extended for large-files

The `fseek()` and `ftell()` interfaces can not be extended to support large files on a 32-bit OS because the POSIX standard defines the use of `long`'s to represent an offset value.

```
long int ftell(FILE *stream);
int fseek(FILE *stream, long int offset, int whence);
```

The use of `long`'s by the standard limits these values to 32-bits on a 32-bit OS.

6.1.1.3 New Non-POSIX API

The following new interfaces have been provided to allow the same functionality as `fseek()` and `ftell()`, but use `off_t`'s instead of `long`'s. These interfaces are provided by both `_LARGEFILE_SOURCE` and `_LARGEFILE64_SOURCE`.

```
#include <stdio.h>
off_t ftello(FILE *stream);
int fseeko(FILE *stream, off_t offset, int whence);
```

Refer to the manpages for specific information about these interfaces.

6.1.1.4 New 64-bit Specific Interfaces and Data types

All of the data types and structures listed in "Section 6.1.1 Affected data types, structures, and API's" have 64-bit counterparts. Similarly, all of the affected POSIX and Non-POSIX API have a new counterpart in the form of *interface*64(). An example of this is open() and open64(). These interfaces are provided for developers who do not want to use the 64-bit compile environment discussed in "Section 6.1.2 Compile Environments". It is not recommended to use these interfaces as it will limit the portability of your application in the future. Refer to the manpages for specifics about these new interfaces. These interfaces are provided by the `_LARGEFILE64_SOURCE` compile option.

New 64-bit Specific Data Types

`blkcnt64_t`

`fsblkcnt64_t`

`fsfilcnt64_t`

`fpos64_t`

`off64_t`

`rlim64_t`

New 64-bit Specific Data Structures

`flock64`

`stat64`

`statvfs64`

`async_request64`

Table 3: New 64-bit Specific Interfaces

creat64()	fgetpos64()	fopen64()	freopen64()	fseeko64()
fsetpos64()	fstat64()	fstatvfs64()	fstatvfsdev64()	ftello64()

Table 3: New 64-bit Specific Interfaces

ftruncate64()	ftw64()	lockf64()	lseek64()	lstat64()
mmap64()	nftw64()	open64()	prealloc64()	stat64()
statvfs64()	statvfsdev64()	tmpfile64()	truncate64()	getrlimit64()
setrlimit64()				

6.1.2 Compile Environments

The implementation of large-files provides two compile environments for developers. These compile environments signify the size of data types and data structures that are affected by large-files. It is important to realize that large-files are supported in both environments, however the interfaces and data types used in each environment are different. Please note that the omission of `_FILE_OFFSET_BITS` is equivalent to `_FILE_OFFSET_BITS=32` for this release. The default of `_FILE_OFFSET_BITS=32` may be changed in future releases of HP-UX.

6.1.2.1 64-bit Compile Environment

The 64-bit compile environment supplies 64-bit versions of the appropriate data types, data structures, and interface calls for a 32-bit application. For instance, in this environment an `off_t` is a 64-bit data type and an `lseek` is a 64-bit `lseek`. It is recommended that large-files applications be built in this environment. To compile a source file in this environment, the following compile flag must be used:

```
-D_FILE_OFFSET_BITS=64
```

An example compilation line for foo.c would look as follows:

```
cc -Ae foo.c -D_FILE_OFFSET_BITS=64 -o foo
```

Beginning with HP-UX 11.0, if compiling a 64-bit application for running on a 64-bit OS, using the 64-bit compile flag, `+DDLP64`, will set `_FILE_OFFSET_BITS` to 64 by default.

6.1.2.2 32-bit Compile Environment

The 32-bit compile environment is the default environment, and is the same environment that has been used in previous releases of HP-UX. Using the same example as above, an `off_t` is a 32-bit data type and an `lseek` is the standard 32-bit `lseek`. Although this compile environment is the default, it can be explicitly requested in the same way as the 64-bit compile environment. The following compile flag may be used:

```
-D_FILE_OFFSET_BITS=32
```

The following examples show how to compile foo.c in the 32-bit compile environment. The first example relies on the default behavior, the second explicitly requests the 32-bit compile environment.

```
cc -Aa foo.c -o foo

cc -Aa foo.c -D_FILE_OFFSET_BITS=32 -o foo
```

6.1.3 Calls and Environments

Note that a 64-bit call is not just a call from the 64-bit environment. Either environment can produce 64-bit calls. In the 32-bit environment, 64-bit calls are produced by the new API. In the 64-bit environment, the POSIX API produces 64-bit calls. The table summarizes this information:

Table 4: Calls produces by an API in a Compile Environment

API	32-bit compile environment	64-bit compile environment
POSIX API	32-bit calls	64-bit calls
New non-POSIX API	32-bit calls	64-bit calls
New 64-bit specific Interfaces	64-bit calls	64-bit calls

Another way of looking at this information is; what combination of compile options produce which API, and are they using 32-bit or 64-bit calls? This information is summarized in the table below. Please note that the omission of `_FILE_OFFSET_BITS` is equivalent to `_FILE_OFFSET_BITS=32`.

Table 5: API made available by compile Flags.

Compile Flags	API Available	Type of Call
_LARGEFILE_SOURCE	New non-POSIX API and POSIX API	32-bit calls
_LARGEFILE_SOURCE and _FILE_OFFSET_BITS=64	New non-POSIX API and POSIX API	64-bit calls
_FILE_OFFSET_BITS=64	POSIX API	64-bit calls
_LARGEFILE64_SOURCE	New API (*64 calls) and New non-POSIX API	New API is 64-bit, other calls are 32-bit. the New non-POSIX API is 32-bit.

6.1.4 Determining if large-files are supported

For developers that want to write code that is portable between systems that support large files and systems that do not support large files, two announcement macros are available. These announcement macros are `_LFS_LARGEFILE` and `_LFS64_LARGEFILE`.

`_LFS_LARGEFILE` will be set to 1 if the system provides the interfaces specified by the `_LARGEFILE_SOURCE` compile option. The following example shows how to query this macro.

```
#if _LFS_LARGEFILE == 1
/** _LARGEFILE_SOURCE is available **/
#else
/** _LARGEFILE_SOURCE not available **/
```

`_LFS64_LARGEFILE` will be set to 1 if the system provides the interfaces specified by the `_LARGEFILE64_SOURCE` compile option. The following example shows how to query this macro.

```
#if _LFS64_LARGEFILE == 1
/** _LARGEFILE64_SOURCE is available **/
#else
/** _LARGEFILE64_SOURCE is not available **/
```

Please note that these macros only identify if the interfaces are available if the source is compiled with the appropriate compile option. The developer must still compile the source with `-D_LARGEFILE_SOURCE` or `-D_LARGEFILE64_SOURCE.`

6.1.5 Importance of Header Files

One last point that can not be over emphasized is the importance of header files to this implementation of large files. All of the mapping of data types, data structures, and interfaces are done in header files. A simplified example of how this mapping is done follows:

```
#if _FILE_OFFSET_BITS == 64
static off_t lseek(a,b,c) off_t b;              { return __lseek64(a,b,c); }
#endif
```

Where you may not have included the required header file to use an interface in the past and gotten away with it, failure to do so with large files will cause incorrect behavior of your application. Unfortunately, the consequences of not including the appropriate header files will not be seen until run time, and in some cases can be destructive. An example of this will be shown in “Section 6.3 Conversion Issues/Examples”.

> ☞ **If the 64-bit compile environment is being used with calls to any of the interfaces listed in Table 2: POSIX API, the appropriate header file must be included in the source file. To determine which header files are necessary, consult the man page for each interface listed in Table 2: POSIX API that is being used in the source file.**

6.2 Conversion Choices

There are basically 5 options for a developer considering large file support:

1. Do nothing. This is a reasonable option if the program does not need to read or write large files, and if an occasional `open` or `stat` failure will not cause confusing behavior.
2. Make the program aware of large files, without actually supporting them. This would involve looking for the new `open` and `stat` errors, and behaving in some sensible way when they are encountered. For example, if your application is a browser which currently exits when `stat` fails, then it might be changed to display something reasonable and continue instead.
3. Use the new API in the 32-bit compile environment.
4. Use the POSIX API and the New non-POSIX API in the 64-bit compile environment. If you do not need a 64-bit version of `ftell()` and `fseek()`, then you do not need the New non-POSIX API.
5. If the application is only for use on a 64-bit OS, compile the application as a 64-bit application.

Options 3, 4 and 5 allow the application to fully support large files. If the application must support large files, one of these three options must be chosen.

6.2.1 Using the New API or POSIX API?

How do you decide between the new API and the POSIX API?

The advantage of the new API is that the type cleanup required can be confined to the routines that handle the files that need to be large. The disadvantage is that source code gets messier, and less portable.

The advantage of the POSIX API is portability. The disadvantage is that you can run into problems mixing objects from the 32-bit and the 64-bit compile environments. There are no problems with linking to libc, because of the header file code that we described above. The linker will not prevent mixing of objects in the same executable, since the object is not in any way marked as being from one environment or the other.

But suppose one of the source files contains a routine that passes an `off_t`, or a `struct stat`, or anything else whose size is affected by the compile environment, to a routine in another file compiled in the other environment. This code won't work, and it can be very difficult figuring out why not, since the source code looks fine.

The problem can also exist between your objects and a library that you use, possibly supplied by some other company. If the library interface contains an `off_t`, or any other data type whose size is changing, then using the POSIX API with the 64-bit environment probably won't work. The library was almost certainly compiled in the 32-bit environment, which means that it expects the smaller data types. It might be just as capable as libc, with the appropriate code in header files to make either environment work, but this is unlikely, and it should not be assumed unless the library supplier says that it works.

Because of these mixing issues, we advise using the POSIX API only when every object file can be compiled in the 64-bit environment. You can mix objects if you're sure that no data is passed around (or referenced as a global) whose type is different in the two environments, but we recommend against it.

6.3 Conversion Issues/Examples

6.3.1 General Conversion Issues

In many programs, assumptions are made that int's, long's, pointers, and offsets are all the same size. This leads to programming practices that are not conducive to converting to large files, and will not be conducive to LP64 data models on 64-bit Operating Systems. Every program that makes the above assumptions will need to be modified to support large files. Typical problems include:

- offsets are assigned to an `int` or a `long`.
- offsets are passed to routines whose parameters are undeclared, or declared as the wrong type.
- constants are passed as offsets without casting in non-ANSI mode. For example:
  - ☞ Existing code fragment
    ```
    lseek(fd, 0, SEEK_SET);
    ```
  - ☞ If we are using the 64-bit compile environment would need to be as follows:
    ```
    lseek(fd, (off_t)0, SEEK_SET);
    ```
  - ☞ If we are using the new 64-bit API
    ```
    lseek64(fd, (off64_t)0, SEEK_SET);
    ```
- precision is lost in shifting. An example:
  - ☞ Existing Code fragment assumes `off_t` same size as `int`.

```
off_t offset;
int blkno;

offset = blkno << BLOCK_SHIFT;
```

☞ In the 64-bit compile environment, the codes should be as follows:

```
off_t offset;
int blkno;

offset = (off_t)blkno << BLOCK_SHIFT;
```

NOTE: The typecasting of blkno forces the promotion of blkno to a 64-bit value before the shift takes place. With out the typecasting, the shift would have occurred on blkno as an int (32-bits), and then the value would have been promoted to 64-bit.

- types are embedded in generic libc calls. For example:

  ☞ Existing printf call.

  ```
  printf("%d\n",offset);
  ```

  ☞ The above code fragment works for a 32-bit offset, but it does not work for a 64-bit offset. In the 64-bit compile environment, offset would require %lld. For portability, use #define for these calls that depend on the value of _FILE_OFFSET_BITS. The above code fragment also assumes that an int and a long are the same size, this may not be a safe assumption in the future.

  ```
  /** Define in header file or top of source **/
  #ifdef _FILE_OFFSET_BITS ==64
  #define offset_format "%lld"
  #else
  #define offset_format "%ld"
  #endif

  /** Somewhere in program **/
  printf(offset_format "\n", offset);
  ```

  ☞ There is also a macro available in <portal.h> that will do this type of mapping. The following code example shows how to use this macro.

  ```
  #include <stdio.h>
  #include <portal.h>
  main()
  {
        off_t a;

        a = 50;

        printf("a is %" PRIdF64 "\n",a);
  }
  ```

  ☞ A similar problem to the above example is the use of strtol() and strtoll(). Calls to these interfaces should be replaced with calls to strtoimax() and strtoumax(). Calls to these interfaces will determine the correct translation for the current environment.

6.3.2 fseek() and ftell() conversion Issues

For programs that use fseek and/or ftell a decision must be made as to how to migrate this code to large files. This source code, even if strict typing had been used, can not be converted by just using the `_FILE_OFFSET_BITS=64` compile option. One option is to change the `ftell()` and `fseek()` calls to `ftello()` and `fseeko()` respectively. The developer would also have to ensure that the appropriate variables are changed from `long int`'s to `off_t`'s. The following two trivial code fragments show what changes would be necessary.

```
long int offset;
int rslt;

offset = ftell(fd);
if(offset != -1)
        rslt = fseek(fd,offset,SEEK_CUR);
```

The above code would need to be changed as follows:

```
off_t offset;
int rslt;

offset = ftello(fd);
if(offset != -1)
        rslt = fseeko(fd,offset,SEEK_CUR);
```

Another option is to use the available macros to determine what types and interfaces are going to be used.The following code demonstrates a way to address this situation and maintain portability.

```
movepos(fd)
FILE *fd;
{
        int rslt;

#if defined(_LFS_LARGEFILE) == 1 && defined(_LARGEFILE_SOURCE)
#define ftell ftello
#define fseek fseeko
#ifdef _FILE_OFFSET_BITS == 64
#define offset_format “%lld”
#else /** _FILE_OFFSET_BITS == 32 **/
#define offset_format “%d”
#endif
        off_t offset;
        off_t num;
#else
#define offset_format “%d”
        long int offset;
        long int num;
#endif

        offset = ftell(fd);
        if(offset != -1)
        {
                num = offset/2;
                rslt = fseek(fd,num,SEEK_CUR);
                if(!rslt)
                printf(“New Position “offset_format“\n”,offset-num);
                else
                printf(“Error!\n”);
        }

}
```

The code fragment above will work on a system that does not support large files and a system that does support large files. It will also work in both the 32-bit compile environment and the 64-bit compile environment.

Note that `ftell()` and `fseek()` have been mapped to `ftello()` and `fseeko()` if the system supports large files and the new non-POSIX API have been enabled. Also note that the `printf()` trick described in the “General Conversion Issues” section is being used in this code.

Note: This is not an issue for 64-bit applications for a 64-bit OS.

6.3.3 Example 1 - Simple conversion

This simple example demonstrates the types of changes that are necessary to existing code. This example is not applicable to 64-bit applications for a 64-bit OS.

6.3.3.1 Original Program

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

print_byte(name,offset)
```

```
char *name;
int offset;
{
   int fd;
   int ret;

   if((fd = open(name,O_RDONLY)) == -1) {
      printf("Open Failed\n");
      return(-2);
   }

   if((ret=lseek(fd, offset, SEEK_SET)) == -1) {
      printf("lseek failed\n");
      return(-3);
   }

   /** Print ret **/
   printf("lseek returned %d\n",ret);
}

save(name)
char *name;
{
   struct stat statb;
   int x;

   if(stat(name,&statb) == -1) {
      printf("Stat Failed\n");
      return(-1);
   }

   x = statb.st_size/2;
   print_byte(name,x);
}

main()
{
   char *name="Fred";
   save(name);
}
```

This program will work fine as long as the file is small. However, if the file is large, the `stat()` call will fail and the program will return a -1.

6.3.3.2 Converted Program using new API

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

print_byte(name,offset)
char *name;
off64_t offset;
{
   int fd;
   off64_t ret;
```

```
   if((fd = open64(name,O_RDONLY)) == -1) {
      printf("Open Failed\n");
      return(-2);
   }

   if((ret=lseek64(fd, offset, SEEK_SET)) == -1) {
      printf("lseek failed\n");
      return(-3);
   }

   /** Print ret **/
   printf("lseek returned %lld\n",ret);
}

save(name)
char *name;
{
   struct stat64 statb;
   off64_t x;

   if(stat64(name,&statb) == -1) {
      printf("Stat Failed\n");
      return(-1);
   }

   x = statb.st_size/2;
   print_byte(name,x);
}

main()
{
   char *name="Fred";
   save(name);
}
```

In this example the `stat()`, `open()` and `lseek()` calls have been changed to the corresponding 64-bit calls. The `statb` structure has been changed to a `struct stat64`, which is like a `struct stat`, except that some fields are larger. One field that's larger is `st_size`, which is now an `off64_t`. Since `st_size` is larger, it must be assigned to a variable of type `off64_t`; hence the change in the declarations of `x` and `offset`. This program will now work on large files as long as it is compiled with `-D_LARGEFILE64_SOURCE`.

6.3.3.3 Converted Code using POSIX API (_FILE_OFFSET_BITS=64)

Now let's look at the converted version using the POSIX API in the 64-bit compile environment. You might think that there is nothing to change, since the existing code already uses the POSIX API, but this isn't true. Even though the program works when compiled in the 32-bit environment, it doesn't work when com piled in the 64-bit environment. That's because the program implicitly assumes that `st_size` can be assigned to an `int` (`x`), and that `x/2` can be passed to a routine that takes an `int` as parameter. These assumptions used to be correct, though unwise. Now they're just wrong.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
```

```
print_byte(name,offset)
char *name;
off_t offset;
{
   int fd;
   off_t ret;

   if((fd = open(name,O_RDONLY)) == -1) {
      printf("Open Failed\n");
      return(-2);
   }

   if((ret=lseek(fd, offset, SEEK_SET)) == -1) {
      printf("lseek failed\n");
      return(-3);
   }

   /** Print ret **/
   printf("lseek returned %lld\n",ret);
}

save(name)
char *name;
{
   struct stat statb;
   off_t x;

   if(stat(name,&statb) == -1) {
      printf("Stat Failed\n");
      return(-1);
   }

   x = statb.st_size/2;
   print_byte(name,x);
}

main()
{
   char *name="Fred";
   save(name);
}
```

This version should look a lot more familiar. In fact, this version could be compiled and run in either the 32-bit or the 64-bit compile environment. The source has no assumptions about whether `off_t` is larger or smaller than an `int` or anything else.

6.3.4 Example 2 - Dealing with Large Files

The following example is a trivial program that will take an existing file, extend it out to 200 bytes, and write "jklmno" at the end of the file. If the file does not exist it creates the file.

6.3.4.1 Original Code

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>


main(argc,argv)
int argc;
char **argv;
{
   int fd;
   off_t rslt;
   ssize_t wrote;
   struct stat statb;

   if(argc < 2) {
      printf("Usage: %s filename\n",argv[0]);
      exit(1);
   }



   /** Check to see if file exists **/
   if(stat(argv[1],&statb) == -1) {
      printf("Stat Failed\n");
      printf("Creating File\n");

      /** Create file **/
      if((fd = creat(argv[1],(mode_t)0777)) == -1) {
         perror("creat fails");
         exit(1);
      }
   }
   else
      if((fd = open(argv[1],O_RDWR)) == -1) {
         perror("open fails");
         exit(1);
      }


   if((rslt=lseek(fd,200,SEEK_SET)) == -1) {
      perror("lseek fails");
      exit(1);
   }

   wrote = write(fd,"jklmno",5);
   close(fd);
}
```

This code works correctly in the 32-bit compile environment and may even appear to be clean for the 64-bit compile environment. However, this example will overwrite the beginning of the file in the 64-bit compile environment if it is compiled in K&R mode. This problem will not arise if the source is compiled in ANSI mode. The existing file "Fred" looked as follows before either program was run.

File Before Either Program is Run

```
This is the beginning of the file. These characters should not
be modified by the second example.
```

Compile line for 32-bit and results of program execution

```
cc example2.c -o example2
```

File after execution

```
This is the beginning of the file. These characters should not
be modified by the second example.
jklmn
```

The result of the 32-bit executable is what we would expect. The characters ”jklmn” have been added to the end of the file.

Compile line for 64-bit and results of program execution

```
cc example2.c -D_FILE_OFFSET_BITS=64 -o example2lf
```

File after execution

```
jklmnis the beginning of the file. These characters should not
be modified by the second example.
```

The result of the 64-bit version may be unexpected as it overwrote the data at the front of the file instead of seeking to the correct position and then writing out the data. What is also distressing is that error checking did not catch this problem. It is very important that constants be typecast as the correct type when being sent as parameters to another function if the source is not being compiled in ANSI mode. The following example will show how to fix this problem.

6.3.4.2 Corrected Source

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>


main(argc,argv)
int argc;
char **argv;
{
   int fd;
   off_t rslt;
   ssize_t wrote;
   struct stat statb;

   if(argc < 2) {
      printf(“Usage: %s filename\n”,argv[0]);
      exit(1);
   }
```

```
    /** Check to see if file exists **/
    if(stat(argv[1],&statb) == -1) {
       printf("Stat Failed\n");
       printf("Creating File\n");

       /** Create file **/
       if((fd = creat(argv[1],(mode_t)0777)) == -1) {
          perror("creat fails");
          exit(1);
       }
    }
    else
       if((fd = open(argv[1],O_RDWR)) == -1) {
          perror("open fails");
          exit(1);
       }


    if((rslt=lseek(fd,(off_t)200,SEEK_SET)) == -1) {
       perror("lseek fails");
       exit(1);
    }

    wrote = write(fd,"jklmno",5);
    close(fd);
}
```

This code now works correctly for both the 32-bit compile environment and the 64-bit compile environment. Notice that the change was simply typecasting the second parameter to `lseek()` as an `off_t`. The line of code that was modified is in bold.

6.3.5 Mixing 32-bit and 64-bit calls

We have already talked about mixing objects from the two compile environments, and advised against it. But two other types of mixing are also possible.

First, the old and new API may be mixed in the same program, and even in the same source file. There is still the chance of an error here (you could pass an `off_t` to a routine expecting an `off64_t`), but at least these problems can be diagnosed by looking at the source.

Second, 32-bit calls and 64-bit calls can be mixed on the same file descriptor. Whether this is a good idea depends on how you do it, and so we look at this issue in some detail here.

You might think that mixing calls on the same file descriptor would have to be done by using both the old and the new API, since we don't recommend linking objects from the two environments. But note that file descriptors can be inherited across `fork` and `exec`, and can also be passed to other processes via special syscalls. That is, a file descriptor open with a 32-bit call might be passed to or inherited by a program that uses 64-bit calls, or vice versa.

One feature of this cross-process mixing is that the process opening the file doesn't always know much about the process on the receiving end. If your application is a shell, and you are asked to redirect output for program X, you usually have no idea if program X is converted to support large files or not. Should you use a 32-bit open or a 64-bit `open`? The answer isn't obvious.

The problems occur when the 64-bit part of the program opens a file and passes the descriptor to the 32-bit part of the program (where the parts may be in the same process, or in different ones). Any other type of mixing should be safe. This means, for example, that a developer should be able to safely use 64-bit calls inside libraries, as long as the library interface doesn't change, and as long as the library does not do a 64-bit `open` and pass the resulting file descriptor back to the calling code. The modified library would work correctly linked to code containing 32-bit calls, or code containing 64-bit calls.

In the more dangerous situation, the 32-bit part of the program (again, in the same process, or in a different process) gets a file descriptor which has been opened with the 64-bit `open`. The file may be large, in which case the program may not `lseek` correctly, possibly because of overflow when calculating the seek offset. If the file is small, the 32-bit part can grow the file to be large.

6.3.5.1 Avoiding Mixing Problems

There is a general strategy for avoiding these mixing problems. Any code that opens a file with the 64-bit `open` should ensure that the file descriptor is *not* passed to code that cannot support large files. For example, a library routine that opens a file for use by the caller (like `fopen`) should rely on some indication from the caller that a 64-bit `open` is required; otherwise, it should do a 32-bit `open`. And shells should use a 32-bit `open` rather than a 64-bit `open` for maximum safety.

Unfortunately, maximum safety has a price. If a shell uses a 32-bit `open`, then it will fail on cases like this:

```
$ app <large_file
```

The solution might be to use a 64-bit `open` after all, which is what our standard shells will do. The argument here is that the application is unlikely to `lseek` on its standard input or output, since these could also be pipes. The argument is not airtight, since the application could check the type of its input and output files, seeking only if they aren't pipes. But it does seem to be true that programs that don't `lseek` are safe, even if they use 32-bit calls on a file descriptor opened with a 64-bit `open`.

6.3.6 Converting Libraries

If your application is a library, then you shouldn't even think about converting the library to support large files in a way that changes the sizes of parameters for existing library entry points. The problem is that people will link your new library with an old application, and the results will be disastrous. Instead, you must add new entry points, as we did for `libc`. This way, existing applications maintain their current behavior. Users of the new entry points must use the new names for them, unless your language allows you to play some naming trick similar to the one we are using for `libc`.

6.4 Common Pitfalls and Debugging Techniques

This section discusses a few key areas that can cause a developer problems, how to recognize the problem, and debugging techniques to resolve these problems.

6.4.1 Not Including Appropriate Header Files

It is common that programs are developed that do not include the appropriate header file for a given interface and these programs execute correctly. Programs such as these will not work in the 64-bit compile environment since the implementation of the 64-bit compile environment is done in header files. The following example demonstrates this point.

☞ **To determine which header files need to be included consult the man page for all interfaces listed in Table 2: POSIX API.**

File: sparse_broken.c

```
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

#define ONEGB 1073741824L

main(argc,argv)
int argc;
char **argv;
{
   int fd;
   off_t offset,rslt;
   ssize_t wrote;
   long val;

   if(argc < 3) {
      printf("Usage: %s num_gbs filename\n",argv[0]);
      exit(1);
   }

   offset = (off_t)ONEGB;
   offset = offset * val;
   printf("offset is %lld\n",offset);


   /** Create file **/
   if((fd = creat(argv[2],(mode_t)0777)) == -1) {
      perror("creat fails");
      exit(1);
   }

   if((rslt=lseek(fd,offset,SEEK_SET)) == -1) {
      perror("lseek fails");
      exit(1);
   }

   wrote = write(fd,"jklmno",5);
   close(fd);
}
```

This program will compile without any errors in both environments. However, if this source is compiled with `_FILE_OFFSET_BITS=64`, the resulting application will not behave correctly. In some cases the application will give a run time error of "Invalid argument", and in other cases the application will just do the wrong thing. Examples of this behavior are shown below:

Compilation line of above source:

```
cc sparse_broken.c -D_FILE_OFFSET_BITS=64 -o sparse_broken
```

Test Runs and resulting Files:

```
sparse_broken 1 lfile1
offset is 1073741824
```

```
lseek fails: Invalid argument
ll lfile1
0 Feb  5 17:59 lfile1



sparse_broken 2 lfile2
offset is 2147483648
lseek fails: Invalid argument
ll lfile2
0 Feb  5 17:59 lfile2



sparse_broken 4 lfile4
offset is 4294967296
ll lfile4
2147377157 Feb  5 17:59 lfile4
```

The size given for lfile4 is incorrect, it should be `4294967301`.

The problem with this source file is that it does not include `<unistd.h>`, which is required for to use the `lseek()` interface. Since `lseek()` is mapped to the 64-bit equivalent in `<unistd.h>`, failure to include it means that the 32-bit `lseek()` is being called. These types of errors can be very troublesome to debug since different run time results occur depending on the input values.

Method to Debug this case

When an application is exhibiting confusing behavior such as this example, the best way to check for header file problems is to use the cc preprocessor. By preprocessing the source file, all of the data and interface mapping that is done for large files will be resolved and you can check the resulting output to ensure that everything is getting mapped as you would expect. The cc preprocessor output can be very verbose, so it is best that you send the output to a file and then examine it's contents. The following example will demonstrate this technique on `sparse_broken.c`.

```
% cc -E sparse_broken.c -D_FILE_OFFSET_BITS=64 > sparse_broken.cpped
% grep lseek sparse_broken.cpped
       if((rslt=lseek(fd,offset,0)) == -1) {
               perror("lseek fails");

%
```

As you can see, there is no mapping of `lseek()` to `__lseek64()`. The following example will show what the cc preprocessor output looks like when the `<unistd.h>` header file is added to this source (sparse.c).

```
% cc -E sparse.c -D_FILE_OFFSET_BITS=64 > sparse.cpped
% grep lseek sparse.cpped
       extern off_t lseek();
extern off_t __lseek64();
static off_t lseek(a,b,c) off_t b;                { return __lseek64(a,b,c); }
        if((rslt=lseek(fd,offset,0)) == -1) {
               perror("lseek fails");
%
```

The important thing to notice in the above code fragment is that `lseek()` is mapped to `__lseek64()` through an in-line function in the header file. This methodology is used in by the 64-bit compile environment to map all of the largefiles interfaces to their 64-bit equivalents.

6.4.2 Not compiling with the correct options

The following example will demonstrate a run time error that will occur if you do not compile with the correct compile options. This example was also chosen because it will demonstrate the error message associated with the new EOVERFLOW error. This program is a very simple program that attempts to open the file sent in as a command line argument. If the file can not be opened, `perror()` is called.

Source: example3.c

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

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

        if(argc < 2) {
                printf("Usage: %s filename\n",argv[0]);
                exit(-1);
        }

        if((fd = open(argv[1],O_RDWR)) == -1) {
                perror("Open Failed");
                exit(-1);
        }

        if((fd = close(fd)) == -1) {
                perror("Close Failed");
                exit(-1);
        }
}
```

Compilation Line:

```
cc example3.c -o example
```

Invocation of example (note: largefile is a 2gb+ file) and results:

```
% example3 largefile
Open Failed: Value too large to be stored in data type
```

This is an important error message to be aware of, as it is a general purpose error message. Developers of large-files applications should have an alarm go off if they receive this error. While this error is used as an example of not compiling the source with the `-D_FILE_OFFSET_BITS=64` compile option, it will also occur if the appropriate header files are not included. In this case, failure to include `<fcntl.h>` and using the value of `O_RDWR` would result in this error.

The correct compilation line is:

```
cc example3.c -D_FILE_OFFSET_BITS=64 -o example3
```

Method to Debug this case

Once again the cc preprocessor is the best means to begin debugging this situation. If `perror()` prints out **"Value too large to be stored in data type"**, the first step is ensure that the application is not execut-

ing the 32-bit interfaces when it should be executing the 64-bit interfaces (this example assumes that you want to correctly handle large files). The same debug steps that applied to example **“Section 6.4.2 Not compiling with the correct options”** apply here.

7. Appendix 1 - HP Specific Implementation Details

This note describes HP-specific aspects of HP's large files implementation. It is intended for readers familiar with the specification entitled *Adding Support for Arbitrary File Sizes to the Single Unix Specification*, from the group calling itself the Large File Summit. That document, and other information about the Summit, is available via http://www.sas.com/standards/large.files/. The comments in this note, and the section number references, are based on the version of March 20, 1996. That is the version submitted by the Summit to X/Open, for possible inclusion in the next version of the XPG.

We describe places where HP differs from the Summit spec, and also supply details of HP's implementation choices where these are left open by the spec. In general, HP supports the entire Summit spec.

Both this note and the Summit spec describe the implementation of large files for the C language only. Other languages are beyond the scope of this document.

data type sizes

HP offers two compile environments for the C language. The first, available without any special flags, has the usual 32-bit file-related data types. In the second environment, the following types are 64 bits:

```
off_t
fpos_t
rlim_t
blkcnt_t
fsblkcnt_t
fsfilcnt_t
```

(Note that `ino_t` is 32 bits in both environments.) The second environment is selected by setting `_FILE_OFFSET_BITS` to 64 (A.3.2.4)

HP also offers the Transitional Extensions (3). All new data types described there are 64 bits on HP machines, with the exception of `dirent64` and `ino64_t`, which are not provided (3.1.2.3).

mixing object files

HP allows linking together of object files from the two compile environments (3.3.4). We ensure that references to libc are resolved properly; for example, "lseek" will automatically refer to the 32-bit `lseek` in the 32-bit compile environment, and to the 64-bit `lseek` in the 64-bit compile environment. Developers must ensure that references that cross object file boundaries do not break if these files are compiled in different environments.

offset maximum

Since `off_t` is 32 bits in the default compile environment, the offset maximum there (1.4, 2.1) is 2 GB - 1. For transitional extensions (3), and in the 64-bit compile environment, the offset maximum is 2**63 - 1 bytes.

file system types

Both HFS (UFS) and JFS support large files. HP's initial large files release, HP-UX 10.20, will not include the V3 protocol for NFS; HP-UX 11.0 does include large file support over NFS using the V3 protocol.

getrlimit and setrlimit

HP-UX maintains resource limits as though they were 32-bit quantities internally. Therefore, resource limits are always representable in an object of type `rlim_t` (2.2.1.19), whether `rlim_t` is 32 bits or 64. `RLIM_SAVED_MAX` and `RLIM_SAVED_CUR` are never returned from `getrlimit` or `getrlimit64`. For `setrlimit` and `setrlimit64`, any limit too large for a 32-bit `rlim_t` is interpreted as `RLIM_INFINITY`.

mmap

Currently, attempts to map past the first 2 GB - 1 bytes with `mmap` in the 64-bit compile environment, or with `mmap64`, fail with `EOVERFLOW`. That is, HP does not support `mmap` on large files. This is expected to change in a future release.

asynchronous I/O

The initial large files release, HP-UX 10.20, does not include support for POSIX Asynchronous I/O at all, on small files or large ones. POSIX Async support is available in HP-UX 11.0, and supports large files.

HP has made available a proprietary async I/O interface for certain ISVs. This interface now supports large files.

lseek

HP returns `EINVAL` for the error case that the Summit designates as `EOVERFLOW` (2.2.1.22), both for historical reasons, and because XPG4 seems to require `EINVAL`.

getconf

HP does not support the new `getconf` macros described in 3.3.4.

mount options

HP supports the mount options mentioned in 3.2.1. Every file system is either a `nolargefiles` system (currently the default), or a `largefiles` file system. This state is set at `mkfs` time, and can be changed with `fsadm`, though `fsadm` cannot convert from `largefiles` to `nolargefiles` if any large files are present. A `mount` with neither option accepts either type of file system. A `mount` invocation that specifies `-o largefiles` or `-o nolargefiles` fails if the file system does not have the type requested. Only HFS (UFS) and JFS support these options, since only these file systems support large files.

A `nolargefiles` file system has no large files on it. Also, none can be created; the system behaves as though the offset maximum for every `open` is 2 GB - 1, regardless of the size of `off_t` or `off64_t`.

shell redirection

Shell redirection is done with large opens (2.3.2).

8. Appendix 2 - Terms/Definitions

Large file

A file which is 2GB's or greater.

Small file

A file which is less than 2 GB.

API

Application Programming Interface.

ABI

Application Binary Interface.

Old API

The POSIX API containing routines such as creat, open, etc.

New API

A 64-bit API containing routines such as creat64, open64, etc.

32-bit compile environment

Compile environment with 32-bit data types (e.g., 32-bit off_t, rlim_t).

64-bit compile environment

Compile environment for a 32-bit application with 64-bit data types (e.g., 64-bit off_t, rlim_t). This compile environment is made available by using the _FILE_OFFSET_BITS compile option. This compile option is used as follows:

```
cc -D _FILE_OFFSET_BITS=64
```

In this document the "64-bit compile environment" is one where file offset and file size related data types become 64-bit.

Old ABI

ABI that is created by 32-bit compile environment.

New ABI

ABI that is created by 64-bit compile environment or 64-bit new API.

32-bit calls

Calls from the old API in the 32-bit compile environment.

64-bit calls

Calls either from the new API or from the 64-bit compile environment.

Large open

An open operation by new API (open64), old API (open) with the O_LARGEFILE flag set, or old API with 64-bit compile environment.

Small open

An open operation by old API (open) with the O_LARGEFILE flag cleared.

_LARGEFILE_SOURCE

This compile option provides two new interfaces, ftello() and fseeko(), to the developer. ftello() and fseeko() will behave just as ftell() and fseek() except, ftello() will return an `off_t` instead of a `long int` and fseeko() will take an `off_t` as it's second parameter instead of a `long int`.

_LARGEFILE64_SOURCE

A compile option in the 32-bit compile environment, this compile option must be specified to use the new API. This compile option will make all of the new 64-bit specific interfaces and data types available to the developer. These interfaces will be in the form of *interface*64(). The 32-bit versions of ftello() and fseeko() are also provided by the _LARGEFILE64_SOURCE compile flag.

_FILE_OFFSET_BITS

This compile option will specify what type of compile environment the developer will use. If the compile option is set to 64, then various data types, structures, and interfaces will be changed to their 64-bit equivalents. If the compile option is set to 32, then these items will be set to 32-bits (32 bits is the default).This compile option is used as `_FILE_OFFSET_BITS=32` or `_FILE_OFFSET_BITS=64`.

MAX_SMALL_FILE

This is a file size constant defined as 0x7fffffff (2 GB -1).