



# CONVEX

Using Large Files  
on SPP-UX

First Edition

 HEWLETT®  
PACKARD



**Hewlett-Packard Company**  
Convex Technology Center  
3000 Waterview Parkway  
P.O. Box 833851  
Richardson, TX 75083-3851  
United States of America



---

# Using Large Files on SPP-UX

---

Order No. DSW-650

First Edition

May 1996

Hewlett-Packard Company  
Convex Technology Center  
Richardson, Texas  
United States of America

---

## Using Large Files on SPP-UX

Order No. DSW-650

© Copyright Hewlett-Packard Company 1996. All Rights Reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

### Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.



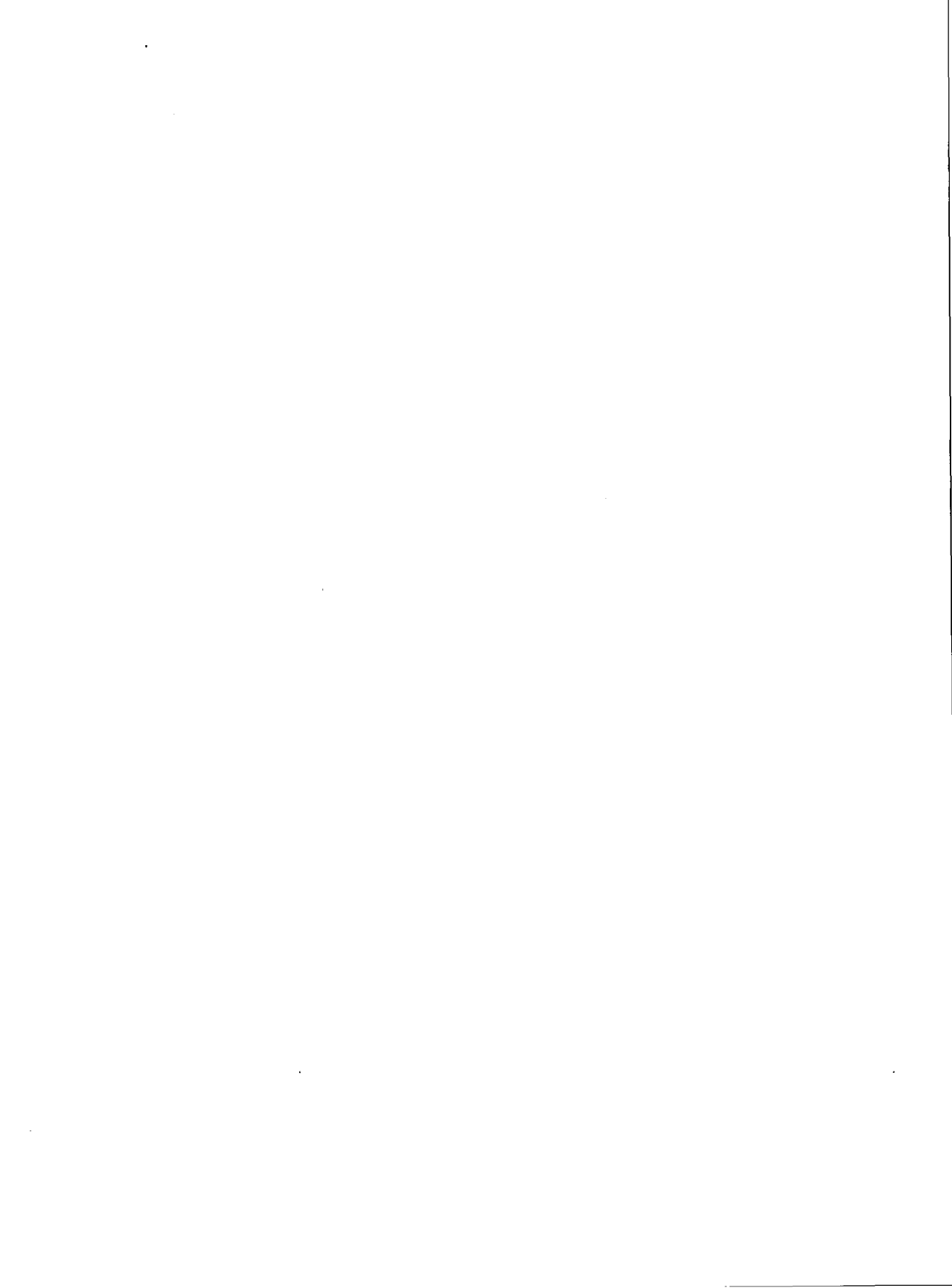
This entire book is recyclable.

Printed in the United States of America

# Revision Information for Using Large Files on SPP-UX

---

Edition	Document No.	Description
First	700-060530-000	Initial release May, 1996.



---

# Contents

---

## How to use this manual. . . . . ix

Purpose and audience . . . . .	ix
Organization . . . . .	ix
Notational conventions . . . . .	x
Notes and cautions . . . . .	x
Associated documents . . . . .	xi
Ordering documents . . . . .	xi
Technical assistance . . . . .	xi

---

## 1 Working with large files . . . . . 1

What are large files? . . . . .	1
File systems and large files . . . . .	2
Determining if your file system supports large files . . . . .	3
File system considerations . . . . .	5
SPP-UX utilities and large files . . . . .	5
Utilities you can use with large files . . . . .	5
Using pipes with large files to work around shell limitations . . . . .	6
Backing up and restoring files . . . . .	7
Detecting and filling holes in large files . . . . .	7
Detecting holes in files . . . . .	8
Filling holes in files . . . . .	8
Preventing hole-filling . . . . .	8
Limitations summary . . . . .	9

---

## 2 Programming with large files . . . . . 11

Programming with large files in Fortran . . . . .	12
Programming with large files in C . . . . .	12
Include file values . . . . .	12
Preparing large files for reading and writing in C . . . . .	13
System calls for use with large files . . . . .	14
Unsupported system calls . . . . .	17
Checking large file support with <code>statfs()</code> . . . . .	18
Using existing programs with large files . . . . .	18



---

# How to use this manual

---

## Purpose and audience

This guide describes large files, how to work with large files, and how to program with large files on Exemplar systems running SPP-UX V3.0 (and later).

The audience for this book is the user who has a working knowledge of UNIX. It is assumed that programmers using this guide are experienced in Fortran or C.

---

## Organization

The first chapter, "Working with Large Files,"

- Defines large files
- Explains how to check if your system supports large files
- Lists the SPP-UX utilities that you can use with large files
- Provides workarounds for shell limitations
- Discusses backing up and restoring large files
- Explains how to detect and fill holes in large files
- Summarizes the conditions to keep in mind when working with large files

The second chapter, "Programming with Large Files,"

- Discusses programming with large files in Fortran
- Discusses programming with large files in C, including:
  - Include file values
  - Preparing large files for reading and writing
  - System calls/limited system calls
  - File system considerations
  - Using existing programs with large files

---

## Notational conventions

This section discusses notational conventions used in this book.

### **Bold monospace**

In command examples, text shown in **bold monospace** identifies user input that must be typed exactly as shown.

### Monospace

In paragraph text, `monospace` identifies command names, system calls, and data structures and types.

In command examples, `monospace` identifies command output, including error messages.

In command syntax diagrams, text shown in `monospace` must be typed exactly as shown.

### *Italic*

In paragraph text, *italic* identifies new and important terms and titles of documents.

In command syntax diagrams, *italic* identifies variables that must be supplied by the user.

---

## Notes and cautions

This document presents notes and cautions in the following formats.

### Note

A Note highlights supplemental information.

---

### Caution

---

A Caution highlights information necessary to avoid damage to the system.

---

## Associated documents

Using large files may require information not specific to the tasks described in this document.

For more information on the SPP-UX operating system or on related software, you can order the books below from Convex Computer Corporation. Note however that the Convex books listed are available in the online document browser PinPoint.

- For more information about the Convex Fortran compiler, refer to the *Fortran User's Guide* (DSW-038).
- For more information about the Convex C compiler, refer to the *C User's Guide* (DSW-086).
- For more information on using SPP-UX, refer to the *HP-UX Reference* (Hewlett-Packard order number B2355-90004).
- For general information on administering SPP-UX, refer to the *SPP-UX System Administration Guide* (DSW-853). (Information in this book pertaining to large files may be incomplete at this time.)

---

## Ordering documents

To order the current edition of this or any other Convex document, send requests to:

Hewlett-Packard Company  
Convex Technology Center  
Customer Service  
P.O. Box 833851  
Richardson, TX 75083-3851

Please include the order number (DSW or DHW number) or the exact title of the document.

---

## Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC) at the following locations:

Within the continental U.S., call 1 (800) 952-0379.

From Canada, call 1 (800) 345-2384.

All other locations, contact the local Convex Technology Center office. You can also use the `contact` utility, if you would like to report any problems you may have with SPP-UX or its associated documentation. For more information refer, to the `contact(1)` man page.



This chapter covers large files concepts and instructions, including:

- What are large files?
- File systems and large files
  - Explains factors of your file system environment that are important when working with large files.
  - Explains how to find out if your file system supports large files.
- SPP-UX utilities and large files
  - Explains conditions that apply to utilities in a large files environment.
  - Discusses limitations of certain functions; information about the limitations is located both in the section where the function is described and in the section “Limitations summary” on page 9.

---

## What are large files?

In SPP-UX, the term *large files* refers to files that are greater than  $2^{31} - 1$  bytes in size (approximately 2 gigabytes). The current upper limit on the size of a large file is one terabyte minus 512 bytes. (One terabyte is  $2^{40}$  bytes.) SPP-UX V3.0 (and later) allows you to create and manipulate large files.

Use the `sysinfo` command to determine what version of SPP-UX you are using.

Some utilities and applications cannot properly process large files. Refer to the section “Utilities you can use with large files” on page 5 for a list of utilities that have been modified to function properly on large files.

---

## File systems and large files

File systems are structures that computers use to organize and store information.

Each file is connected or related to another file, starting with the root (/) directory file and continuing down through an unlimited number of files and directories.

The term *file system* can refer to the entire file hierarchy or to a subsection of the file tree. In this document, the term refers to a subsection of the file tree (a collection of files, directories, and file management structures that is assigned by your system manager to a certain portion of the disk system).

In SPP-UX, the file system is hierarchical, as illustrated in Figure 1.

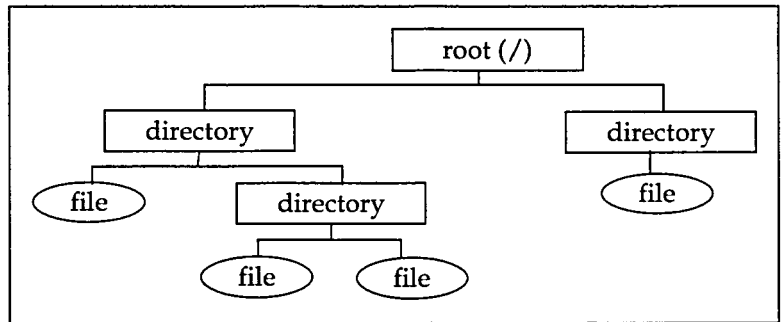


Figure 1 File system hierarchy

---

## Determining if your file system supports large files

Before you create or manipulate a large file, make sure your file system supports large files by making sure

1. You are *not* in an NFS-mounted file system.
2. The file system has the magic number/featurebit pair indicating support for large files.

The details of these steps are given below.

**Step 1** Check that you are *not* working in an NFS-mounted file system.

Large files are not supported on NFS file systems. The currently supported NFS protocol limits the size of files you can access via NFS to two gigabytes or less.

The Network File System (NFS) allows you to mount file systems on various machines enabling you to access files across all the mounted file systems. You access normal (that is, nonlarge) files in NFS-mounted file systems just like you access files that really are on a particular machine.

To find out if you are working in an NFS-mounted file system, use the `bdf` command to report file system information:

```
% bdf .
```

File system	kbytes	used	avail	capacity	Mounted on
/dev/st3	4140152	2980928	745232	80%	/test

If the "File system" column of the `bdf` output starts with `/dev/xxx` (where `xxx` represents a three-character string), you are in a system that can support large files. Thus, in the example above, the file system does support large files, assuming the file system has the correct magic number/featurebit pair.

```
% bdf .
```

File system	kbytes	used	avail	capacity	Mounted on
fantasia:/home	249138	102512	121712	46%	/rmt/fantasia/home

If, however, the "File system" column contains *machinename*, you are working in a file system that is NFS-mounted from remote host *machinename*. If this is the case, you cannot create or manipulate large files in this file system. In the example above the machine *fantasia* is listed in the "File System" column indicating that large files are not supported on *fantasia*.

**Step 2** Determine if the file system has the appropriate magic number/featurebit pair (indicating support for large files) by using the `cnx_dumpefs` command as shown below. See the `cnx_dumpefs(1M)` man page for more information. Use the `statfs()` system call to check for large file support from within a C or Fortran program. Refer to "Checking large file support with `statfs()`."

- a. Enter

```
% cnx_dumpfs -l file_system_mount_point
```

where *file\_system\_mount\_point* is the directory given in the "Mounted on" column in the `bdf` output.

Sample output follows:

```
magic    FD_MAGIC_2  
featurebits  FSF_LARGEFILES
```

- b. Verify that the magic number is `FD_MAGIC_2` and that the `featurebits` field contains `FSF_LARGEFILES`.

If the magic number and feature bits fields do not indicate that large files are supported on your file system, contact your system administrator.

---

## File system considerations

Although the stated limit for large files is currently one terabyte minus 512 bytes, some combinations of the items

- Block size
- Number of free blocks

can make it impossible to write a file that large. On file systems with block sizes less than 32 kilobytes, SPP-UX uses triple-indirect block pointers to write large files at the maximum size. In such a file system (with block size less than 32 kilobytes), the availability of block pointers determines the maximum file size you may write.

Use the `dumpfs` command to determine the block size on your system. See the `dumpfs(1M)` man page for more information.

The optimal block size is 64 kilobytes; this size produces the greatest number of block pointers.

Consult your system administrator if you are interested in having a system's block size changed.

---

## SPP-UX utilities and large files

Some SPP-UX utilities work on large files in the same manner as they do on regular files. Other utilities place restrictions on large file operations. The following sections explain how some of the SPP-UX utilities interact with large files.

---

### Utilities you can use with large files

Not all utilities are practical for use with large files. For example, you are not likely to try to examine a large file using `more` because the size would make viewing the file unmanageable and make finding anything in it difficult. The SPP-UX utilities most useful with large files have been modified to work with large files. These utilities are:

- `cat`
- `chmod`
- `cp`
- `dd`
- `dump`
- `dumpfs`, `cnx_dumpfs`
- `find`
- `fbackup`
- `frecover`

- fsck
- fsckclean
- fsirand
- ftp
- ls
- mkfs, cnx\_mkfs
- mount
- mv
- ncheck
- newfs, cnx\_newfs
- restore
- rcp
- rm
- tail

Use of all utilities is subject to the file system restrictions described in the section “File systems and large files.”

## Note

**Do not use any utility to place a large file in a file system where it is not allowed—your file may be truncated.**

For example, suppose you want to move large file `test1` from file system `/foo` to file system `/bar` using `mv`. If `/bar` cannot hold large files, using the command `mv /foo/test1 /bar` returns a write failed message. `/bar/test1` contains only the first two gigabytes of `test1`, and the remaining data is lost.

---

### Using pipes with large files to work around shell limitations

The `sh` and `csh` shells and their variants cannot directly manipulate large files. This means that functions such as shell redirection do not work for large files.

You can use pipes with large files. Using pipes in combination with various utilities, you can replicate many shell functions as shown in Table 1.

The second row of the table demonstrates how you can use pipes to feed large files to utilities that otherwise could not handle them. This method works with most utilities.

**Table 1** Using pipes to work around shell limitations

Limitation	Use	In place of
Using the shell to write large files	<code>cat fname1   dd of=fname2</code>	<code>cat fname1 &gt; fname2</code>
Using utilities that must be fed large files through pipes	<code>cat fname   grep pattern</code>	<code>grep pattern fname</code>

For more information on shells, please refer to Hewlett-Packard's *Using HP-UX* and *A Beginner's Guide to HP-UX*, each of which contains several chapters on how to use shells.

---

## Backing up and restoring files

You can use:

- `dump`
- `restore`
- `fbackup`
- `frecover`

to backup and restore file systems. Use these utilities with large files in the same manner as you use them with regular files.

Three other file system utilities that have limited usefulness with large files are:

- `tar` (these commands do not work
- `pax` for files greater than
- `cpio` eight gigabytes in size)

These utilities have a limit of 11 octal digits built into their format; you can use them to back up files up to eight gigabytes in size.

If any of these utilities (`tar`, `pax`, `cpio`) encounters a file larger than eight gigabytes, it issues a warning and skips that file.

---

## Detecting and filling holes in large files

A *hole* is a region of a file that has not been written to but has been bypassed with the `lseek()` or `lseek64()` system call. Knowing about the holes in your files (and how to maintain or fill these holes) allows you to use disk space more effectively.

A hole is created, for example, when a new file is opened. The first 512 kilobytes (0 through 511) are skipped, then a string is written starting at kilobyte 512. The presence of the hole is recorded, but

its content is not stored on disk, and it does not occupy any disk space.

A large file that contains a hole may fit on a particular file system, whereas a file of the same size without the hole may not fit. The utilities `cp` and `mv` fill in holes under some circumstances. The following sections explain how you can detect and fill holes.

### Detecting holes in files

You can detect holes by comparing the output from `ls` and from `du`.

Suppose, for example, that the file `myfile` has a size of 10 megabytes as reported by `ls` but is only taking up 32 kilobytes of actual disk space, according to `du`:

```
% ls -l myfile
-rw----- 1 joe 10485760 Sep 10 15:26 myfile
% du myfile
32      myfile
```

This discrepancy occurs because `myfile` contains one or more holes.

### Filling holes in files

If you copy or move the file `myfile` to another directory, the holes are filled with zeros, and disk usage corresponds to actual file size:

```
% cp myfile bigfile.with.holes.filled
% ls -l bigfile.with.holes.filled
-rw----- 1 joe 10485760 Sep 10 15:30
bigfile.with.holes.filled
% du -a bigfile.with.holes.filled
10240   bigfile.with.holes.filled
```

The next section, “Preventing hole-filling,” explains how to maintain holes when copying files.

### Preventing hole-filling

Use `cp` with the `-z` option to preserve the holes in large files (and consequently, reduce disk usage) when copying:

```
% cp -z myfile bigfile.with.holes.preserved
```

---

## Limitations summary

The following list summarizes the conditions you should keep in mind when working with large files:

- Not all utilities and applications can properly process large files.
- The file system's magic number must be `FD_MAGIC_2`, and the system's `featurebits` field must contain `FSF_LARGEFILES` on *all* file systems where you are using large files.
- The currently supported NFS protocol limits the size of files you can access via NFS to at most two gigabytes.
- The `sh` and `cs` shells cannot directly manipulate large files; however, there are workarounds.
- You may unknowingly fill holes in large files when copying or moving files. (Hole-filling can result in inefficient use of disk space.)
- If `tar`, `pax`, or `cpio` encounters a file larger than eight gigabytes, it issues a warning and skips that file.
- If you attempt to use any utility to place a large file in a file system where it is not allowed, your file may be truncated.



This chapter covers:

- Fortran support for large files
- C support for large files:
  - Include file values
  - System calls
  - Unsupported system calls
- Checking large file support from within a program
- File system considerations
- Using large files with existing programs

---

## Programming with large files in Fortran

Convex Fortran V9.3 (and later) supports large files—large formatted and unformatted files can be read and written. The Fortran I/O interface extends transparently to large files.

Direct access files are limited to  $2^{31}-1$  RECORDS. The actual size of such a file is then limited by the size of individual records. This restriction is due to the use of the INTEGER\*4 data type for the following specifiers:

- REC= specifier in READ and WRITE statements for direct access files
- NEXTREC= specifier in INQUIRE statements

You must relink Fortran programs with Convex Fortran V9.3 (or later) in order to read and write large files. All Fortran programs linked with the Fortran V9.3 (and later) libraries have large file access. If you do not have access to Convex Fortran V9.3 (and later), contact your system administrator.

---

## Programming with large files in C

Convex C V6.3 (and later) supports large files—large formatted and unformatted files can be read and written. The C I/O interface requires you to specify certain include file values and system calls to properly handle large files. The following sections

- List and describe the needed include files.
- Describe how to prepare large files for reading and writing.
- List and describe the necessary system calls.

---

### Include file values

The following include files contain definitions and prototypes that are useful in programming with large files:

<sys/cnx\_fcntl.h>

Defines the O\_LARGEFILE flag, struct flock64, F\_GETLK64, F\_SETLK64, and F\_SETLKW64; includes prototypes of creat64() and open64().

<sys/cnx\_fs.h>

Defines new magic number of a file system that allows large files (FD\_MAGIC\_2); defines the large file feature bit (FSF\_LARGEFILES).

<sys/cnx\_ino.h>

Redefines di\_size to be correct for an inode from a large-file-aware file system.

<sys/ino.h> must be included before <sys/cnx\_ino.h>

<sys/cnx\_inode.h>

Defines the inode structure that is used on a large-file-aware file system.

<sys/inode.h> should not be included with

<sys/cnx\_inode.h>;

(the only difference between the two files is in the way `ic_size` is defined)

<sys/cnx\_stat.h>

Defines `struct stat64`; includes prototypes for `stat64()`, `fstat64()`, and `lstat64()`.

<sys/cnx\_types.h>

Defines `off64_t`, `fblkcnt64_t`, and `fsblkcnt64_t`; `off64_t`, which is 64 bits long, defines possible offsets (and maximum size) of a file.

<sys/cnx\_unistd.h>

Includes prototypes for `lseek64()`, `ftruncate64()`, and `truncate64()`.

---

## Preparing large files for reading and writing in C

When programming with large files, you have to set the `O_LARGEFILE` (or large file) flag to allow a program to read or write beyond the two-gigabyte boundary in a file.

Use one the options below to set the `O_LARGEFILE` flag.

- Set the `O_LARGEFILE` flag when opening a file, as shown in the following examples:

```
fd=open("largefilename", O_RDWR | O_LARGEFILE);
```

or

```
fd=fopen("largefilename", "rwl");
```

or

```
fd=open64("largefilename", O_RDWR);
```

- Set the `O_LARGEFILE` flag by using `fcntl()`:

```
fd=open("largefilename", O_RDWR);
```

```
flag=fcntl(fd, F_GETFL, 0);
```

```
fcntl(fd, F_SETFL, flag|FLARGEFILE);
```

You must use `fcntl` properly when working with large files; shortcut methods may lose the `O_LARGEFILE` flag. For example, the following call structure loses the `O_LARGEFILE` flag:

```
fcntl(fd, F_SETFL, FASIO);
```

```
/* Do not use the call structure above */
```

However, the call structure below maintains the large file flag setting:

```
flag=fcntl(fd, F_GETFL, 0);
flag|=FASIO;
fcntl(fd, F_SETFL, flag);
```

---

## System calls for use with large files

The following system calls are available for use with large files:

- `creat64()`
- `fcntl()`
- `fgetpos64()`
- `fopen()`
- `fseek64()`
- `fsetpos64()`
- `fstat64()`
- `ftell64()`
- `ftruncate()`
- `ftruncate64()`
- `lseek()`
- `lseek64()`
- `lstat64()`
- `open()`
- `open64()`
- `read()`
- `stat64()`
- `truncate()`
- `truncate64()`
- `write()`

Each function with a name ending in 64 (indicating the function is a 64-bit function) has a 32-bit counterpart whose name is the same except for the ending 64 (`lseek64()`, `lseek()`; `fseek64()`, `fseek()`; etc.). Except as noted below, the 64-bit function has the same functionality as its 32-bit counterpart, but takes a 64-bit offset.

The list below describes the behavior of various system calls with respect to large files. These system calls and libraries are located in the static library `/usr/convex/all/spp1/libc.a`.

`creat()` (Unsupported for large files)

Use `creat64()` in place of `creat()`.

The `creat()` system call is equivalent to

```
open(path, O_CREAT|O_TRUNC|O_WRONLY, mode);
```

Note that this equivalence causes `creat()` to fail if the file referenced by *path* exists and is a large file.

`creat64()`

The `creat64()` library call is equivalent to

```
open(path, O_CREAT|O_TRUNC|O_WRONLY|O_LARGEFILE, mode);
```

Unlike the `creat()` system call, `creat64()` successfully truncates and opens large files.

`fcntl()`

Use the `fcntl()` system call to set or clear the `O_LARGEFILE` flag using the `F_SETFL` command. Use the `F_GETFL` command to get the current state of the `O_LARGEFILE` flag and of other flags you can set or clear.

`fcntl()` allows the `O_LARGEFILE` flag to be set and cleared regardless of the size of the open file.

`fgetpos64()`

Get stream position as a 64-bit offset.

`ftruncate()`, `ftruncate64()`

To use `ftruncate()` successfully on large files, make sure the `O_LARGEFILE` flag is set. If the flag is clear, the operation fails and returns -1 with `errno` set to `E_OVERFLOW`.

`ftruncate64()` is the same as `ftruncate()`, but uses a 64-bit offset. Like `ftruncate()`, `ftruncate64()` requires the `O_LARGEFILE` flag to be set in order to truncate a large file.

`fseek64()`

Seek on a stream using 64-bit offsets. Implemented with `lseek64()`.

`fsetpos64()`

Set stream position as a 64-bit offset.

`ftell64()`

Returns stream position as a 64-bit offset.

`lseek()`, `lseek64()`

The `lseek64()` system call behaves like the `lseek()` system call except that the offset argument and returned value are defined as type `off64_t` instead of `off_t`. This allows

`lseek64()` to take file offsets larger than `lseek()` and return any resulting large values.

If the resulting offset of an `lseek()` operation is legal (that is, less than the maximum file size), but too large to be represented in the return value (for example, a signed 32-bit integer for `lseek()`), the kernel returns -1 and sets `errno` to `EOVERFLOW`.

`open()`

The `open()` system call fails to open a large file unless the `O_LARGEFILE` flag is specified in the flags field. In addition to the normal errors, the `open()` system call returns -1 and sets `errno` to `EOVERFLOW` if the file specified by the *path* argument exists, is a large file, and `O_LARGEFILE` was not specified.

When `O_TRUNC` is specified without `O_LARGEFILE`, `open` does not truncate a large file, but returns -1 and sets `errno` to `EOVERFLOW`.

`open64()`

Open a file, with `O_LARGEFILE` bit implied.

`read()`

Regardless of the maximum file size—which is determined by the setting of the `O_LARGEFILE` flag (see `fcntl()`)—the `read()` system call does not read data at a file offset greater than the size of the file. In this case, -1 is returned and `errno` is set to `EINVAL`.

To illustrate this idea, consider the following three cases:

- If the current file offset is equal to the size of the file, a read requesting at least 1 byte of data returns a value of 0 (no bytes read) and leaves `errno` unchanged.
- If the current file offset is greater than the size of the file, a read requesting at least 1 byte of data returns -1 and sets `errno` to `EINVAL`.
- If the size of a file is greater than or equal to 2 gigabytes and the `O_LARGEFILE` flag is clear, then the `read()` system call behaves as if the file was 2 gigabytes - 1 byte in size, and therefore returns `EINVAL`.

`stat()`, `lstat()`, `fstat()` (Unsupported for large files)

Use `stat64()`, `lstat64()`, and `fstat64()` for large files in place of these calls.

The `stat()`, `lstat()`, and `fstat()` system calls fail if the specified file is a large file (larger than 2 gigabytes - 1 byte in size), because the size of the file cannot be represented in a signed 32-bit integer. In this case, `stat()`, `lstat()`, and `fstat()` returns -1 and sets `errno` to `EOVERFLOW`.

`stat64()`, `lstat64()`, `fstat64()`

The `stat64()`, `lstat64()` and `fstat64()` system calls use the `stat64` structure instead of the `stat` structure to return information about a file. The main difference between these two structures is that the `st_size` field in the `stat64` structure is defined to be of type `off64_t` instead of `off_t`.

This allows the `stat64()`, `lstat64()` and `fstat64()` system calls to return information for files larger than 2 gigabytes - 1 byte in size.

Other than this difference, `stat64()`, `lstat64()` and `fstat64()` behave exactly as their 32-bit counterparts.

`truncate()`, `truncate64()`

The `truncate()` system call truncates large files. The `truncate64()` system call redefines the length argument to be an `off64_t`.

`write()`

If a write requests that more bytes be written than there is room (for example, the maximum file size is exceeded or the storage medium is full), only as many bytes as there is room are written. For example, if there is only room for 20 bytes and a write requests that 512 bytes be written, only 20 bytes are written.

If the current file offset is equal to or greater than the maximum file size as determined by the setting of the `O_LARGEFILE` flag (see `fcntl()`), a write requesting at least 1 byte of data returns -1 and sets `errno` to `EOVERFLOW`.

---

## Unsupported system calls

Because they require 32-bit file offsets, the following system calls do not work with large files:

- `mmap()`
- `getrlimit()`
- `setrlimit()`

These system calls do not adversely affect large files, but their usefulness with respect to large files is limited:

`mmap()` maps only the first two gigabytes of a file.

`setrlimit()` can limit file size with byte granularity only up to two gigabytes (which is less than the size of a large file). It may set the file size limit to `RLIM_INFINITY` to allow access to any size file.

Do not use `setrlimit` or `getrlimit`; `setrlimit()` cannot set, and `getrlimit()` cannot process, limits above two gigabytes.

---

## Checking large file support with `statfs()`

Use `statfs()` to check large-file support from within a program.

Verify that in the output from `statfs()`, the magic number (`f_magic`) is `FD_MAGIC_2` and that the `FSF_LARGFILES` bit (`f_featurebits`) is set. Refer to the `statfs(2)` man page for more information.

If the magic number and feature bits fields do not indicate that large files are supported on your file system, contact the system administrator for the computer in question.

---

## Using existing programs with large files

The behavior of existing programs with large files is determined by the behavior of the system calls those programs use. Recompiling an old program in a large file environment has no effect because it still uses the same system calls.

The danger in using an existing program with a large file is to the file, not the program. A program that does not understand large files cannot open a large file.

The following list explains how several system calls behave if they encounter a large file, assuming the programs that contain the system calls have not been modified in any manner that may facilitate working with large files.

See the section “System calls for use with large files” for information on system calls that work properly with large files.

`lseek()`

Fails on a large file if the file offset is already greater than two gigabytes or if a successful seek would cause the offset to be greater than two gigabytes. It ignores the `O_LARGEFILE` bit.

`open()`

Fails when attempting to open a large file.

`stat()`

Fails when performed on a large file.

`truncate()`, `ftruncate()`

Truncates a large file at the two-gigabyte boundary.

These limitations also apply if a program that understands large files system calls another program that does not.

---

# Index

---

## F

file systems  
  considerations when programming with large files 5  
FORTRAN support for large files 12  
fseek64 command with large files 15  
ftell64 command with large files 15

---

## G

getrlimit command with large files 17

---

## I

include files for programming with large files 5

---

## L

large files  
  definition 1  
  exemptions 3  
  FORTRAN support 12  
  system calls with 14  
  utilities that can be used with large files 5

---

## M

mmap command with large files 17

---

## P

pipes, using with large files 6  
programming with large files  
  include files 5  
  using existing programs 18

---

## S

setrlimit command with large files 17  
shell restrictions with large files 6  
system calls with large files 14

---

## U

utilities  
  that can be used with large files 5







HEWLETT®  
PACKARD

CONVEX  
PRESS

ORDER NUMBER  
DSW-650

DOCUMENT NUMBER  
700-060530-000

