

<b>1. Introduction .....</b>	<b>1</b>
<b>2. Hard Disk Structure .....</b>	<b>3</b>
2.1 Physical Disk Access: Cylinders, Heads and Sectors .....	3
2.2 Hard Disk Partitions and Logical Sectors .....	4
<b>3. The Second Extended File System.....</b>	<b>6</b>
3.1 A Brief History of Linux Filesystems .....	6
3.2 Common File System Concepts .....	7
3.2.1 Blocks .....	7
3.2.2 Inodes .....	8
3.2.3 Directories.....	8
3.3 The Structure of the Second Extended File System.....	9
3.3.1 Block Groups: The Building Blocks of the Filesystem.....	10
3.3.2 The Superblock.....	12
3.3.3 The Group Descriptors.....	14
3.3.4 The Block and Inode Bitmaps .....	15
3.3.5 Inodes .....	16
3.3.6 Directories.....	19
3.3.7 How the Second Extended Filesystem Works: A Mini Case Study.....	20
<b>4. Ext2lib Design .....</b>	<b>21</b>
4.1 The Multi-layer Design of Ext2lib .....	21
4.2 Layer 0 - The Hardware Access Layer .....	21
4.3 Layer 1 - The Ext2 Translation Layer .....	23
4.3.1 A Mini-case study: Obtaining a Directory Listing.....	24
4.4 Layer 2 - The Application Interface Layer.....	26
4.4.1 The part command.....	26
4.4.2 The mount command .....	26
4.4.3 The ls command .....	27
4.4.4 The cd command .....	27
4.4.5 The cp command .....	27
4.4.6 The get inode command .....	28
<b>5. Ext2lib Implementation.....</b>	<b>29</b>
5.1 The Structure of the Source Code.....	29
5.2 The implementation of layer 0: The Hardware access layer.....	30
5.3 Non-DOS Hard Disk Access Under MS-DOS: BIOS Disk Functions.....	30
5.4 Using Interrupt 0x13 Under MS Windows 95: DPMI.....	31
5.5 Layer 0: The SimulateRMInt Function .....	32
5.6 Layer 0: The ReadPhysicalSector Function .....	33
5.7 Layer 0: The GetDiskParams Function.....	34
5.8 Layer 0: The ReadPartInfo Function .....	35
5.9 Layer 0: The GetPartTable Function .....	37
5.10 Layer 0: The ReadLogicalSector Function .....	38
5.11 Layer 1: The ReadBlock Function .....	38
5.11.1 The Read-Ahead Cache .....	39
5.11.2 The FIFO Cache .....	39
5.12 Layer 1: The ReadSuperBlock and GetSuperBlock Functions.....	40
5.13 Layer 1: The ReadGroupDesc and GetGroupDesc Functions .....	41

5.14 Layer 1: The GetlNode Function .....	41
5.15 Layer 1: The GetBlockList Function .....	42
5.16 Layer 1: The ReadDir Function .....	44
5.17 Layer 1: The PathToInode Function .....	45
5.18 Layer 1: The CopyFile Funciton.....	46
5.19 Layer 1: The GetPartInfoNice Function .....	48
5.20 Layer 2: Notes .....	49
5.21 Layer 2: The get_part_info Function.....	49
5.22 Layer 2: The ext2_mount_fs Function .....	49
5.23 Layer 2: The ext2_cd Function .....	50
5.24 Layer 2: The ext2_ls Function .....	51
5.25 Layer 2: The ext2_get_inode Function .....	52
5.26 Layer 2: The ext2_cp Function .....	52
5.27 The Cmdlin User Interface.....	53
<b>6. The Cmdlin User Interface.....</b>	<b>54</b>
<b>7. Analysis.....</b>	<b>56</b>
<b>8. Conclusions.....</b>	<b>58</b>
<b>9. References.....</b>	<b>60</b>
<b>10. Bibliography .....</b>	<b>60</b>

## Appendices

**Appendix A - Original Project Proposal**

**Appendix B - Initial Project Specification**

**Appendix C - Plan of Action**

**Appendix D - Cmdlin Manual**

**Appendix E - Ext2lib Programmers Implementation Guide**

## 1. Introduction

The use of the Linux operating system is rapidly spreading amongst home and commercial users alike. Often Microsoft Windows users migrating to Linux install it on a separate disk partition, employing a dual boot system to enable the user to specify which operating system to use at system start-up. Linux is equipped with the capability of reading the filesystem that is used by Windows, however Windows exclusively reads its own filesystem. This presents a considerable inconvenience to users in that files located on a Linux partition cannot be accessed when Windows is being used. To access a file on the Linux filesystem, the user must re-boot the computer, start Linux, and copy the file to the Windows partition. The computer then has to again be re-booted to start Windows and use the file. It would, therefore, be a considerable advantage if files on the Linux partition could be accessed from within Windows.

It is on developing an application that achieves this task that this project is based.

Linux uses a filesystem called the Second Extended Filesystem (ext2), and Windows 95 uses a filesystem known as VFAT. The VFAT filesystem is essentially the same as the FAT (File Allocation Table) filesystem used by previous versions of Windows and MS-DOS, except that it allows the use of long filenames. The two filesystems are not compatible and MS-DOS or Windows does not even recognise the existence of a partition that contains an ext2 filesystem. It is for these reasons of overcoming fundamental incompatibility and the direct usefulness of the application that makes this an interesting and technically challenging project.

To ensure maximum flexibility, both in development and implementation, the functions used to access the ext2 filesystem have been incorporated into a Windows dynamically linked library (EXT2LIB.DLL). This approach was chosen because it separates the user interface from the 'nuts and bolts' of accessing the filesystem. This means that the library can be used as a component when developing an application that accesses the ext2 filesystem. Alongside the library, a Windows application, Cmdlin, has been developed that uses the library to provide a simple command line style user interface. This allows the user to view partition information for the hard disks connected to the PC, mount a selected ext2 partition, and then use familiar 'cd' and 'ls' commands to browse the directory structure contained in the filesystem and a 'cp' command to copy files from the ext2 to the FAT partition. This user interface has been designed to provide an environment that will be familiar to most computer users and is simple enough to serve as a useful tool for demonstrating the implementation and use of the ext2lib library. The cmdlin user interface and the ext2lib library together make up an application that performs the task of reading an ext2 filesystem from within Windows.

The ext2lib library provides a 'virtual command line' to the calling application. This is to say that it exports functions for displaying partition information, mounting a partition, ls, cd and cp. The ls function writes the directory listing that would normally be expected at a command line into memory, where it can be used by the calling application. The cd and cp commands perform their respective operations, changing the current directory and copying a file in the way that would be expected at a real command line, also writing any output into memory for the use of the calling application.

The next two sections of this dissertation give additional background material on hard disk structure and the Second Extended filesystem. The information contained in these sections is vital to understanding how ext2lib was implemented.

A high level account of how the ext2lib library was designed and a detailed guide to how this design was implemented with a brief look at the cmdlin user interface follow.

The final two sections give an analysis of the success of the project and the conclusions.

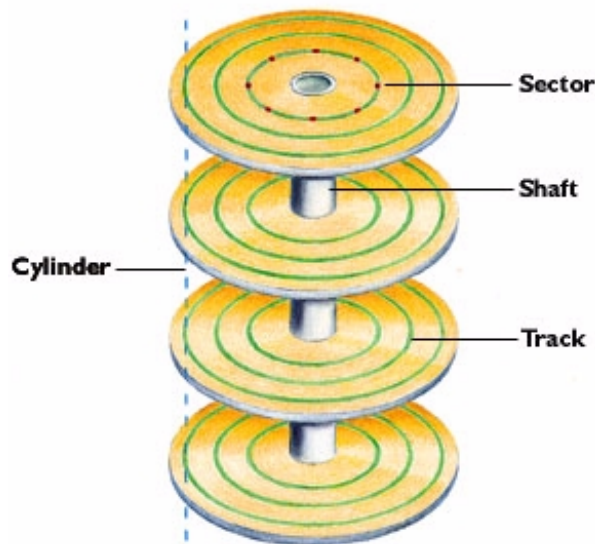
The appendices give the original project proposal and specification, a user manual for the cmdlin interface and a programmers implementation manual for the ext2lib library.

## 2. Hard Disk Structure

This section aims to outline several concepts related to hard disk access that are necessary to understand the low level disk access routines that provide the foundations for accessing any filesystem on disk.

### 2.1 Physical Disk Access: Cylinders, Heads and Sectors

Information stored on a disk is divided into small parts (usually 512 bytes for hard disks) called sectors. When a disk I/O operation is carried out this is the smallest amount of information that can be read or written in one go. These sectors are known as physical sectors. A hard disk consists of one or more



platters on which information is stored. These are magnetically coated disks stacked one on top of another on a central rotating shaft. These platters are read by a head suspended a fraction of a millimetre away from both sides of each platter. The head is free to move across the platter from its centre to its outer edge. The sectors are arranged in concentric circles called tracks on both sides of the platters. A cylinder describes all tracks at a given head position across all platters.

A three figure reference is used to uniquely identify each sector on a disk: a cylinder number, head number and sector number (CHS). The cylinder number gives the cylinder to be read, and the head number gives the head, it can be seen that these two numbers are sufficient to position the correct head above the desired cylinder. The sector number then gives the individual sector to be read from the disk. It should be noted that although these are referred to as physical sectors, the cylinder, head and sector numbers no longer relate to the physical position of the sector on modern hard disks, as internal translation is carried out to overcome certain design limitations.

So, to conclude, reading or writing a physical sector on a disk involves specifying its CHS co-ordinates. This level of disk access pays no attention to what is actually on the disk, this must be done by the operating system or user program.

## 2.2 Hard Disk Partitions and Logical Sectors

Hard disks are often divided up into different sections called partitions. Each logical partition can behave like a hard drive in itself. For example, in MS-DOS, a disk can be divided into two partitions, the primary and extended partitions. Logical drives can then be defined in the extended partition, each of which has its own drive letter and behaves like a separate disk.

A hard disk can be divided into a maximum of four partitions. The information about these partitions is held in a special part of the disk called the partition table. It is located at Cylinder 0, Head 0, Sector 1 on the disk. The structure of a partition table entry is shown below:

```
struct partition {
    unsigned char boot_ind;
    unsigned char head;
    unsigned char sector;
    unsigned char cyl;
    unsigned char sys_ind;
    unsigned char end_head;
    unsigned char end_sector;
    unsigned char end_cyl;
    unsigned short start_sectlo;
    unsigned short start_secthi;
    unsigned short nr_sectsl;
    unsigned short nr_sectshi;
};
```

Each partition on a disk has an entry of the form shown above in the partition table, the function of each of the values is as follows:

- `boot_ind` - This indicates whether the partition can be booted from. A value of `0x80` indicates that the partition is bootable.
- `head`, `sector` and `cyl` - These make up the CHS location of the start of the partition.
- `sys_ind` - This is a hex value that represents the type of partition to which the entry refers, a few relevant types are:

```
0x05 - DOS extended partition.
0x06 - DOS primary partition.
0x82 - Linux swapfile partition
0x83 - Linux filesystem partition.
```

- `end_head`, `end_sector` and `end_cyl` - These make up the CHS location of the end of the partition.
- `start_sectlo` and `start_secthi` - The low and high byte of the start sector, with sectors numbered from zero upwards.
- `nr_sectsl` and `nr_sectshi` - The low and high byte of the total number of sectors that make up the partition.

Within a partition the individual sectors are located by a single number, known as the 'logical sector' number. The CHS reference could still be used, but the single figure is easier to handle. The numbering starts from zero with the first sector of the partition, this sector is known as the 'boot sector' of the partition as it contains a small piece of code used to start the operating system.

Logical sector numbers are obtained by numbering each sector sequentially starting from track 0, head 0, sector 1 (logical sector 0) of the partition and continuing along the same head, then to the next head until the last sector on the last head of the track is counted. Thus, logical sector 1 is track 0, head 0, sector 2; logical sector 2 is track 0, head 0, sector 3; and so on. Numbering then continues with sector 1 on head 0 of the next track.

Converting the logical sector number of a partition to an absolute CHS reference is a two stage process, firstly the offset of the start of the partition (`start_sect`) has to be added to the logical sector number, this gives an absolute logical sector number. This number is then converted to a CHS reference. As explained in the previous section, the sector can now be read from disk.

### 3. The Second Extended File System

References: [2] [3] [4]

#### 3.1 A Brief History of Linux Filesystems

Linux was originally developed as an extension to the Minix operating system and, therefore, the only filesystem to be supported was the Minix filesystem. This had two major limitations: the Minix filesystem stores block addresses as 16 bit integers so the filesystem size is limited to 64 Megabytes, and the directories contain fixed size entries with the maximum filename length of 14 characters. Due to these limitations work began on adding support for new filesystems into the Linux kernel.

To allow new filesystems to be added more easily into the Linux kernel, an additional layer, the Virtual File System (VFS) layer was developed. This layer sits between the system calls interface and the actual filesystem code to provide an indirection layer which handles the file oriented system calls and calls the necessary functions in the filesystem code.

Once the VFS had been implemented, a new filesystem, 'The Extended File System' was implemented and added to Linux 0.96c. This new filesystem was largely based on the Minix filesystem, but removed the two major limitations, for the Extended File System the maximum partition size was 2 Gigabytes and the maximum file name length was 255 characters. The Extended File System still had some unresolved problems though: There was no support for inode modification and data modification timestamps, separate access and the filesystem used linked lists to keep a record of free blocks and inodes and this produced poor performance and the lists became unsorted leading to filesystem fragmentation.

To resolve these problems, in January 1993 two new filesystems were released in their alpha stage,. The Xia filesystem and the Second Extended Filesystem (Ext2fs). The Xia system was heavily based on the Minix filesystem kernel code and offered few improvements over this system. Ext2fs was based on the Extended Filesystem code with many improvements and offered significant advantages, not least that it was designed with evolution in mind and contains space for many future improvements. Ext2fs is now very stable and has become the filesystem of choice for most Linux users.

## **3.2 Common File System Concepts**

All filesystems are simply made up of blocks of data on a disk. This information needs to be organised in a way that is not only robust and reliable, but also provides good I/O performance. This is the job of the filesystem. The concepts described in this section are common to many different filesystems and provide a basis for the more detailed discussion of the second extended filesystem that follows. All Linux filesystems implement a set of common concepts that are derived from the Unix operating system: Individual files are represented by inodes, which store information about the file and hold the locations of the data blocks that make up the file and directories are implemented as a special type of file containing a list of directory entries.

### 3.2.1 Blocks

As mentioned above, all filesystems are divided into logical 'blocks' of data, each read or write from disk uses an integral number of these blocks. A block is the smallest amount of space that can be allocated to a file. These blocks are known as 'logical blocks', that is they are a group of physical disk sectors. The logical block size must be the physical sector size multiplied by a power of two.

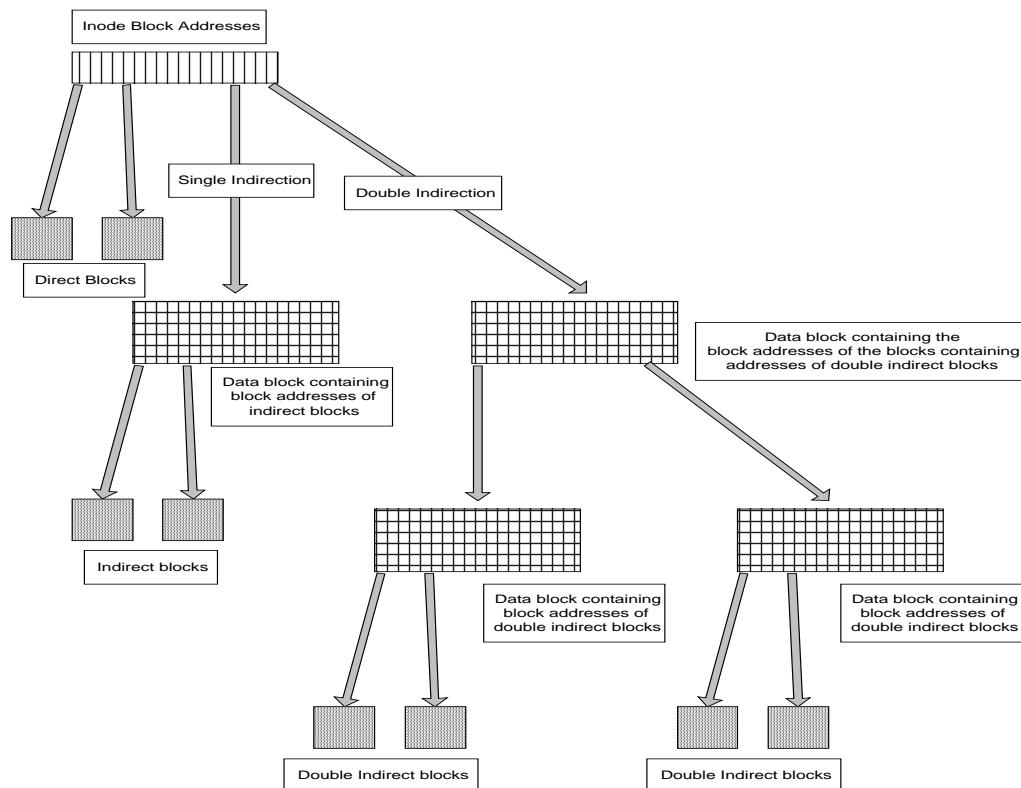
If a file is smaller than a block, or the end of the file falls between block boundaries, the extra space in the block is wasted. This wastage can be alleviated by using smaller allocation units known as fragments, however these are not implemented in all filesystems yet.

The logical blocks are numbered from zero up to the total number of blocks minus one, this number is known as the block address, it is used to identify a unique block within the filesystem.

These blocks may be organised into groups of blocks to make the allocation of blocks and management of the filesystem housekeeping information easier and more robust.

### 3.2.2 Inodes

Each file is represented by a structure on disk called an inode, Each inode contains the file type, access rights, owners, timestamps, size and pointers to data blocks. The addresses of data blocks allocated to a file are stored in its inode. Some block addresses are stored on the inode itself, these are known as 'direct blocks', whilst further data block addresses are stored in a data block pointed to from the inode, these are called 'indirect blocks'. Similarly additional data block addresses are stored as 'double indirect blocks': A block of indirect block addresses pointed to from the inode. Triple indirect blocks are implemented in a similar way, with an extra layer of indirection added. The diagram below illustrates this concept.



### 3.2.3 Directories

Directories are structured in a tree like hierarchy with each directory containing files and subdirectories. Directories are implemented as a special type of file, with an inode to identify it. This file contains a list of the file and subdirectory names and their corresponding inode number. When a pathname is used it must be converted by the kernel to an inode number by following the directory tree from the root directory, through any subdirectories, to actual file or

subdirectory itself. The inode can then be read from disk and subsequent file operations performed.

### **3.3 The Structure of the Second Extended File System**

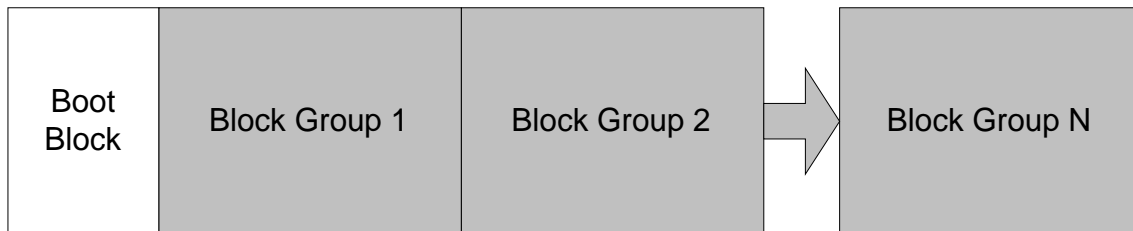
This section intends to provide a reference to the second extended filesystem, its layout on disk and how this fits together to produce a working filesystem. The section starts by outlining the different data structures stored on disk and then shows how these are used to manipulate files. The focus of the analysis will be on the aspects of the filesystem that are involved in read only access, as it is these areas that are currently implemented in this project.

The second extended filesystem implements the standard unix file types: regular files, directories, device special files and symbolic links. It can handle extremely big partitions, up to 4 Terrabytes with recent work done on the VFS layer. The long filename support in ext2fs currently limits the filename length to 255 characters, but this could be extended to 1012 if necessary. In addition to these standard features, ext2fs implements many advanced features such as :

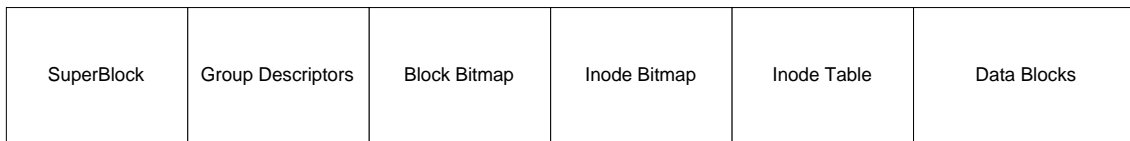
- File attributes allow the kernel behaviour, when acting on a set of files, to be modified by the user. Attributes can be set on a file or directory. In the case of directory attributes, all files within that directory inherit the attributes.
- The logical block size used throughout the filesystem can be modified when the filesystem is created, sizes are typically 1024, 2048 or 4096 bytes.
- Ext2fs keeps track of the filesystem state, a special flag (in the superblock) is used by the kernel code to indicate the status of the filesystem. When an ext2 filesystem is mounted in read/write mode its state is set to 'not clean', only if the filesystem is unmounted correctly is the flag set back to 'clean'. This allows the kernel to determine whether a filesystem is likely to contain errors when attempting to mount it so checks can be performed.

### 3.3.1 Block Groups: The Building Blocks of the Filesystem

The blocks that make up the filesystem are organised into groups known as block groups. These groups allow an order to be imposed on block allocation within the filesystem, and provide a useful backup mechanism for important filesystem housekeeping information. They are laid out as shown in the diagram below:



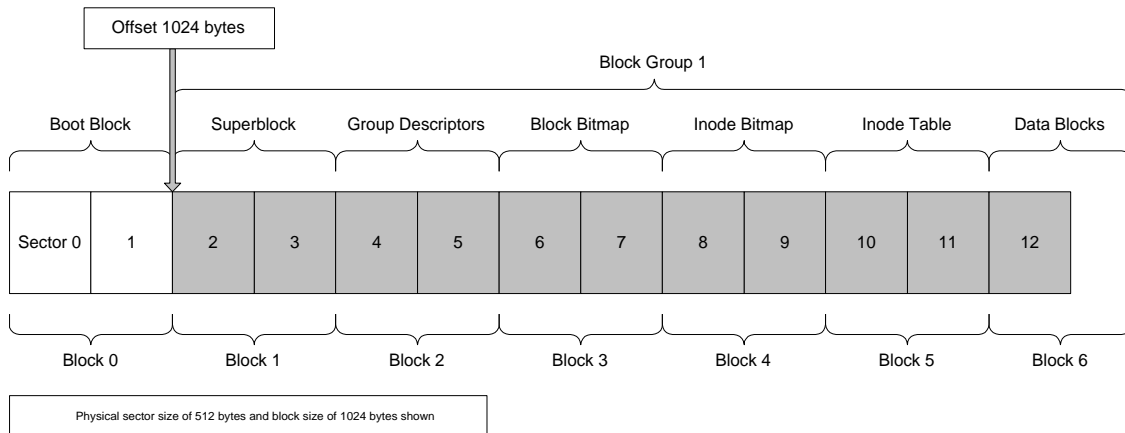
Each block group contains a copy of crucial filesystem information, the superblock and the group descriptors. Only the copy located in block group 1 is used by the kernel but the others provide a backup. In addition to this each block group also contains a piece of the filesystem: a block bitmap, an inode bitmap, a section of the inode table and the actual data blocks. These are laid out on the disk in the order shown in the diagram below:



- The Superblock is a structure that contains vital information used by the kernel when mounting and accessing the filesystem.
- The Group Descriptors hold information relating to each block group in the filesystem. Each group descriptor only relates to the particular group it describes. They are located one after another in the same order as the block groups on the disk that they describe.
- The Block and Inode bitmaps are used to indicate which blocks/inodes within the group have been allocated.
- The Inode table contains all the inodes for this group.
- The data blocks make up the rest of the group and are where files and directories are actually stored.

Having the inode table and the data blocks that are allocated from those inodes physically close on disk provides performance benefits due to reduction in disk head seeks during file I/O operations.

The block groups are located on the disk one after another, the numbering and correspondence of the logical blocks with the logical disk sectors is shown below.



The first group, and hence the copy of the superblock and group descriptors that the filesystem uses is always located at offset 1024 from the beginning of the partition. Subsequent groups follow as described above.

### 3.3.2 The Superblock

The superblock is a structure found at the beginning of each block group, it contains information vital to mounting and accessing the filesystem. Currently the superblock size is 1024 bytes. The structure of the superblock, shown below, is defined in `ext2_fs.h`.

```
/*  
 * Structure of the super block  
 */  
struct ext2_super_block {  
    unsigned long   s_inodes_count;  
    unsigned long   s_blocks_count;  
    unsigned long   s_r_blocks_count  
    unsigned long   s_free_blocks_count  
    unsigned long   s_free_inodes_count  
    unsigned long   s_first_data_block  
    unsigned long   s_log_block_size  
    long           s_log_frag_size;  
    unsigned long   s_blocks_per_group  
    unsigned long   s_frags_per_group  
    unsigned long   s_inodes_per_group  
    unsigned long   s_mtime;  
    unsigned long   s_wtime;  
    unsigned short  s_mnt_count;  
    short          s_max_mnt_count;  
    unsigned short  s_magic;  
    unsigned short  s_state;  
    unsigned short  s_errors;  
    unsigned short  s_pad;  
    unsigned long   s_lastcheck;  
    unsigned long   s_checkinterval;  
    unsigned long   s_creator_os;  
    unsigned long   s_rev_level;  
    unsigned short  s_def_resuid;  
    unsigned short  s_def_resgid;  
    unsigned long   s_reserved[235];  
};
```

- `s_inodes_count` - The total number of inodes on the filesystem.
- `s_blocks_count` - The total number of blocks on the filesystem.
- `s_r_blocks_count` - The total number of blocks reserved for use by the superuser.
- `s_free_blocks_count` - The total number of free blocks on the filesystem.
- `s_free_inodes_count` - The total number of free inodes on the filesystem.
- `s_first_data_block` - The position on the filesystem of the first data block. Usually this is block 1 for a filesystem with a block size of 1024 bytes, and 0 for larger block sizes.
- `s_log_block_size` - This is used to calculate the logical block size (in bytes). The actual block size is given by:  $1024 \ll s\_log\_block\_size$ .

- `s_log_frag_size` - This is used to calculate the fragment size (in bytes). The actual fragment size is given by: `1024 << s_log_frag_size` if `s_log_frag_size` is positive and `1024 >> -s_log_frag_size` if `s_log_frag_size` is negative.
- `s_blocks_per_group` - The total number of blocks per group.
- `s_frags_per_group` - The total number of fragments per group.
- `s_mtime` - The time when the filesystem was last mounted.
- `s_wtime` - The time when the superblock was last written.
- `s_mnt_count` - The number of times the filesystem has been mounted in read-write mode without being checked.
- `s_max_mnt_count` - The number of times the filesystem can be mounted in read-write mode before a check is forced.
- `s_magic` - The superblock magic number, used for identification purposes. For a normal ext2 filesystem the magic number is `0xEF53`, on older filesystems (prior to version 0.2b) it is `0xEF51`.
- `s_state` - The current state of the filesystem, it contains either of two values: `EXT2_VALID_FS (0x0001)` for a clean filesystem, or `EXT2_ERROR_FS (0x0002)` for a filesystem that was not unmounted cleanly.
- `s_errors` - Indicates which operation to perform when errors are detected.
- `s_pad` - Unused padding.
- `s_lastcheck` - The time when the last check was made on the filesystem.
- `s_checkinterval` - The maximum possible time between checks on the filesystem.
- `s_reserved` - Unused.

All times are measured in seconds since 00:00:00 GMT, January 1<sup>st</sup> 1970.

On mounting a filesystem, the first operation is to read the superblock and perform checks for the validity of the filesystem. As can be seen from the descriptions above, the superblock contains information that is fundamental to the operation of the filesystem, it is for this reason that spare copies are made at the beginning of each block group.

### 3.3.3 The Group Descriptors

A group descriptor exists for every group on the filesystem, it gives information about the location of the group on disk and the free space within the group. A copy of all the group descriptors for the filesystem is stored in each block group, after the superblock. The group descriptors simply follow one after another, in the order that the groups they refer to are located on disk. The structure of a group descriptor is shown below, taken from `ext2_fs.h`.

```
/*
 * Structure of a blocks group descriptor
 */

struct ext2_group_desc
{
    unsigned long   bg_block_bitmap;
    unsigned long   bg_inode_bitmap;
    unsigned long   bg_inode_table;
    unsigned short  bg_free_blocks_count;
    unsigned short  bg_free_inodes_count;
    unsigned short  bg_used_dirs_count;
    unsigned short  bg_pad;
    unsigned long   bg_reserved[3];
};
```

- `bg_block_bitmap` - The block address of the block containing the block bitmap for the group.
- `bg_inode_bitmap` - The block address of the block containing the inode bitmap for the group.
- `bg_inode_table` - The block address of the first block of the inode table for the group.
- `bg_free_blocks_count` - The number of free blocks within the group.
- `bg_free_inodes_count` - The number of free inodes within the group.
- `bg_used_dirs_count` - The number of inodes allocated to directories within the group.
- `bg_pad` - Padding.
- `bg_reserved` - Not Used.

The group descriptors are used by the kernel for locating a particular group on disk and for balancing the allocation of blocks between the groups on a filesystem. The kernel only uses the first copy of the group descriptors, the others are for backup purposes only.

### 3.3.4 The Block and Inode Bitmaps

The filesystem uses these bitmaps to keep track of allocated blocks and inodes within each group on the filesystem. The block and inode bitmaps for each group are located at the beginning of each group, after the superblock and group descriptors. Each bit in the bitmap refers to a block or inode within the group, a value of 1 indicates that the block or inode has been allocated, and 0 indicates that it is free. The bits are ordered from the first inode or block in the group to the last. To examine the allocation status of a specific block or inode, the group that it belongs to first has to be identified. Once this has been done, the bit for the block or inode is located within the block or inode bitmap for that group. Its allocation status can then be determined.

### 3.3.5 Inodes

An inode uniquely describes a file or directory. It's main purpose is to hold the locations of the actual data blocks making up the file. The inodes are stored one after another in the inode table, a section of which is stored in each group. When allocating blocks for a particular file, the kernel will generally allocate blocks from the same group that contains the inode for the file, as performance is enhanced by having the inode and the blocks it points to located nearby on the disk. A number is used to individually address each inode, starting at 0 for the first inode and its value simply following on from one group to the next, through all the groups in the filesystem. The group containing a particular inode can then be quickly determined by simply knowing how many inodes are contained in each group (this is a fixed number, recorded in the superblock).

The structure of an inode on disk is shown below, taken from `ext2_fs.h`:

```
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
    unsigned short i_mode;
    unsigned short i_uid;
    unsigned long i_size;
    unsigned long i_atime;
    unsigned long i_ctime;
    unsigned long i_mtime;
    unsigned long i_dtime;
    unsigned short i_gid;
    unsigned short i_links_count;
    unsigned long i_blocks;
    unsigned long i_flags;
    unsigned long l_i_reserved1;
    unsigned long i_block[EXT2_N_BLOCKS];
    unsigned long i_version;
    unsigned long i_file_acl;
    unsigned long i_dir_acl;
    unsigned long i_faddr;
    unsigned char l_i_frag;
    unsigned char l_i_fsize;
    unsigned short i_pad1;
    unsigned long l_i_reserved2[2];
};
```

- `i_mode` - The type of file (character, block, link, etc.) and access rights to the file. This field is best described by representing it as an octal number. Since it is a 16 bit number, there will be 6 octal digits. The rightmost four digits are bitwise fields:

The last three digits (Octal digits 0,1 and 2) are the file permissions, in the form `rwxxrwxrwx`. Digit 2 refers to the user, digit 1 to the group and digit 0 to everyone else.

The leftmost two octal digits are used to indicate the type of file that the inode points to:

Note that the leftmost octal digit can only take the value of 0 or 1, since the total number of bits is 16. The type of file is one of:

- 01 - FIFO file.
- 02 - Character device.
- 03 - Directory
- 06 - Block Device
- 10 - Regular File
- 12 - Symbolic Link
- 14 - Socket

- `i_uid` - The user id of the owner of the file.
- `i_size` - The file size in bytes.
- `i_atime` - The time the file was last accessed.
- `i_ctime` - The time the inode information for the file was last changed.
- `i_mtime` - The time the files content was last modified.
- `i_dtime` - The time the file was deleted.
- `i_gid` - The group id of the file
- `i_links_count` - The number of links pointing to the file.
- `i_flags` - This takes one or more of the following values:

`EXT2_SECRM_FL 0x0001` - Secure deletion. If this flag is set, when the file is deleted random data is written in its place.

`EXT2_UNRM_FL 0x0002` - Undelete. If this flag is set and the file is deleted, the system must store enough information for the file to be recovered (provided the space taken by the file has not been overwritten).

`EXT2_COMPR_FL 0x0004` Compress file. The content of the file is compressed. The filesystem code must use compression/decompression algorithms when accessing the file.

`EXT2_SYNC_FL 0x0005` - Synchronous Updates - The inode and indirect blocks are to be written to synchronously only.

Not all of the above features are implemented in the current version of `ext2fs`.

- `i_reserved1` - Not used.
- `i_block[ EXT2_N_BLOCK ]` - Pointers to blocks allocated to the file represented by this inode.

The inode contains `EXT2_N_BLOCKS`, this is currently 15. Of these block addresses, the first `EXT2_NDIR_BLOCKS` (currently 12) are direct pointers to data blocks. The following entry points to a block of pointers to data

blocks (indirect blocks). The entry after points to a block of pointers to blocks of pointers to data blocks (double indirect blocks) and the final entry points to a block of pointers to blocks of pointers to blocks of pointers to data blocks (triple indirect blocks).

- `i_version` - The version of the file, used by NFS.
- `i_file_acl` - The Access control list of the file (not used).
- `i_dir_acl` - The Access control list for the directory (not used).
- `i_faddr` - The block where the fragment of the file resides.
- `i_frag` - The number of fragments in the block.
- `i_size` - The size of the fragment.
- `i_pad1` - padding
- `i_reserved2` - Not used.

Fragments are allocation units smaller than blocks that can be used to alleviate the problems of wasted, or slack, space at the end of files. They are not fully implemented yet.

### 3.3.6 Directories

Directories are implemented as a special kind of file. The file contains a linked list of directory entries. A linked list is used instead of fixed size entries to save disk space. The directory entry links a filename with an inode number, it also stores the name length and entry length so a particular entry can be located within the list. The filename is not stored as a null terminated string, hence its length has to be stored.

The structure of a directory entry is shown below, taken from `ext2_fs.h`:

```
/*
 * Structure of a directory entry
 */
#define EXT2_NAME_LEN 255

struct ext2_dir_entry {
    unsigned long    inode;
    unsigned short  rec_len;
    unsigned short  name_len;
    char            name[EXT2_NAME_LEN];
};
```

- `inode` - The inode number of the file.
- `rec_len` - the directory entry length.
- `name_len` - the filename length.
- `name` - the name of the file.

An entry exists in the directory for each file contained within that directory. The first two entries in any directory are `‘.’` and `‘..’`, which point to the current directory and the parent directory respectively.

### 3.3.7 How the Second Extended Filesystem Works: A Mini Case Study

To illustrate how the kernel, or another application, uses the structures defined above to navigate a filesystem we shall examine the case study of reading a text file. Suppose we want to read the file `‘/home/dave/test.txt’`. The following steps are necessary to read the file:

1. The filesystem must be mounted. The superblock information and the group descriptors are read from disk and checked for validity.
2. Locating the inode for any file begins at the root directory. This has a special inode number, defined by `EXT2_ROOT_INO` in `ext2_fs.h`, currently 2.
3. This inode is located in the first group, the group descriptor is read and the location of the block address of the inode table found.
4. The inode is located and read into memory. To determine which group an inode is in, a simple calculation can be done. The group is given by:  
`Integer_Part(<Inode Number> / s_inodes_per_group)`
5. The blocks pointed to by the inode contain the directory file. These are read, in effect reading the directory listing into memory.
6. A search through the entries in the root directory is carried out for the filename ‘home’. Once it is found it’s inode can be read into memory. This inode points to the file containing the directory listing for the ‘home’ directory. As with the root directory, the listing is read into memory.
7. The ‘home’ directory listing is searched for the entry named ‘dave’. Once this is found, as before, its inode and the directory file pointed to by the inode are read into memory.
8. The directory listing now in memory is for the directory named ‘dave’. It is searched for the entry with name ‘test.txt’. Once this is found, just as with the directories, it’s inode is read into memory.
9. This inode points to all the blocks making up the file ‘test.txt’. These can now be read, thus reading the file.

Steps 2 to 8 from a process known as ‘path to inode resolution’.

## 4. Ext2lib Design

### 4.1 The Multi-layer Design of Ext2lib

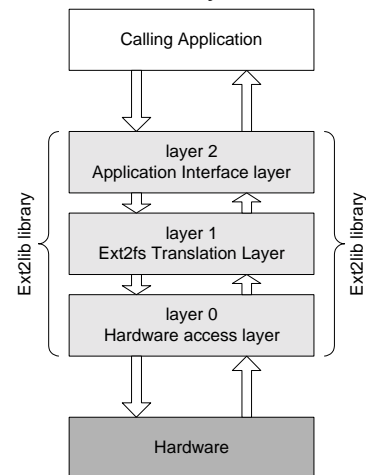
In designing ext2lib, it seemed appropriate to divide the construction into three distinct layers:

- Layer 0 - The hardware access layer
- Layer 1 - The ext2fs translation layer
- Layer 2 - The application interface layer

The reason for this is mainly one of safety. The functions in layer 0 could potentially cause hard disk corruption, as they perform access to the hard disk at a low level. This design provides a level of abstraction between the interface to the calling application and any functions that could compromise the integrity of a hard disk. The functions that are exported from the library are all contained in layer 2, the application interface layer, and any functions that perform low level access to disk hardware are in layer 0. So there is no way that the calling application can execute possibly damaging functions.

Layer 1 holds all the functions that are necessary to convert the raw data read from the disk by functions in layer 0 into useful filesystem information such as directory listings and files. This information is then used by functions in layer 2 to implement the basic filesystem commands ls, cd and cp. It is these commands that will form the basis of layer 2. Layer 2 is to provide what can be thought of as a 'command line' that can be used by the calling application to perform operations on the ext2 filesystem.

This three layer structure also provides modularity within the program code, which makes it easier to update and maintain, and therefore more stable.

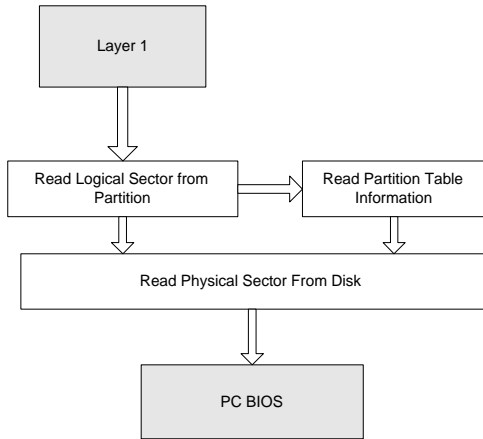


### 4.2 Layer 0 - The Hardware Access Layer

The function of layer 0 is to provide layer 1 with a function that can read a logical sector from a specified drive and partition. To achieve this, layer 0 has to be able to:

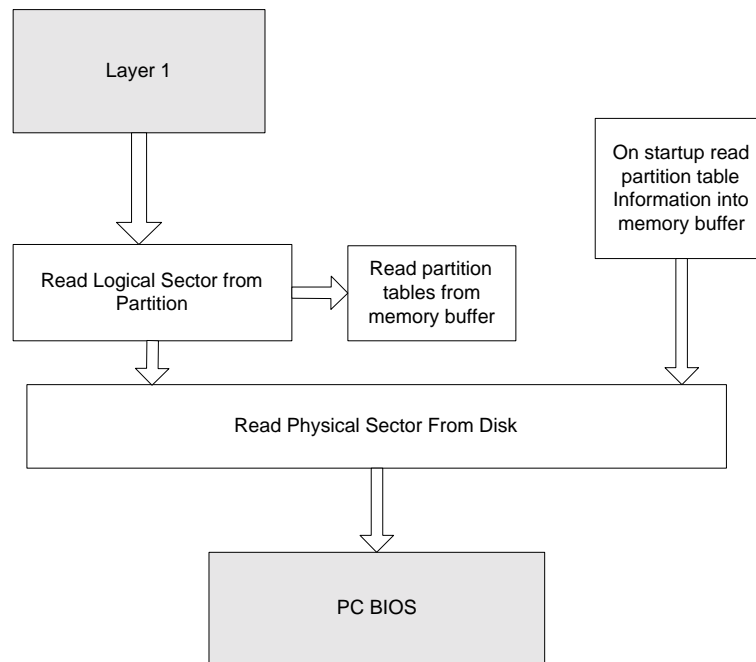
- Read a physical sector from any disk connected to the PC.
- Read and process the partition tables from any disk connected to the PC.

The task that layer 0 has to perform forms the base of the whole application, as reading from the hard disk forms the basis of any operation that the ext2lib library will have to perform. It is essential, therefore, that the mechanism that is used is both reliable and efficient. The layer needs to consist of three distinct functions: A function to read a physical sector from disk, a function to read the partition tables from disk, and a function that can be called by functions in layer 1 to read a logical sector from a partition. It is this function alone that provides the interface to layer 1 from layer 0.



The diagram on the left shows this arrangement. This provides a reliable structure for layer 0, it is, however, not the most efficient. The partition tables are extremely unlikely to be altered whilst ext2lib is in operation, so it is inefficient to read them from disk each time they are needed by the read logical sector function. The partition tables are therefore read into memory when ext2lib is first loaded, and are buffered there until ext2lib is unloaded. Any time that the read logical sector function need access them, they are stored in memory and do not have to be read form disk, this provides a great performance increase. This forms the structure of layer 0, and is illustrated below

The diagram on the left shows this arrangement. This provides a reliable structure for layer 0, it is, however, not the most efficient. The partition tables are extremely unlikely to be altered whilst ext2lib is in operation, so it is inefficient to read them from disk each time they are needed by the read logical sector function. The partition tables are therefore read into memory when ext2lib is first loaded, and are buffered there until ext2lib is unloaded. Any time that the read logical sector function need access them, they are stored in memory and do not have to be read form disk, this provides a great performance increase. This forms the structure of layer 0, and is illustrated below



**The Structure of Layer 0**

### 4.3 Layer 1 - The Ext2 Translation Layer

This layer forms the bulk of the library. It must process the raw data read using the read logical sector function provided by layer 0 to produce output such as inodes and directory listings that can be used by layer 2 to implement basic filesystem commands such as ls and cd. It must also perform the task of copying files

The tasks that the functions in this layer have to perform are:

- Reading a logical block from the filesystem
- Reading the superblock information
- Reading a specified group descriptor
- Reading a specified inode
- Reading the list of blocks allocated to a particular inode
- Converting a path name to an inode number
- Copying a file from the ext2 filesystem to the DOS filesystem
- Reading a directory listing

The interface that layer 1 has to provide to layer 2 consists of the functions to read a directory, convert a path name to an inode number, read an inode and copy a file.

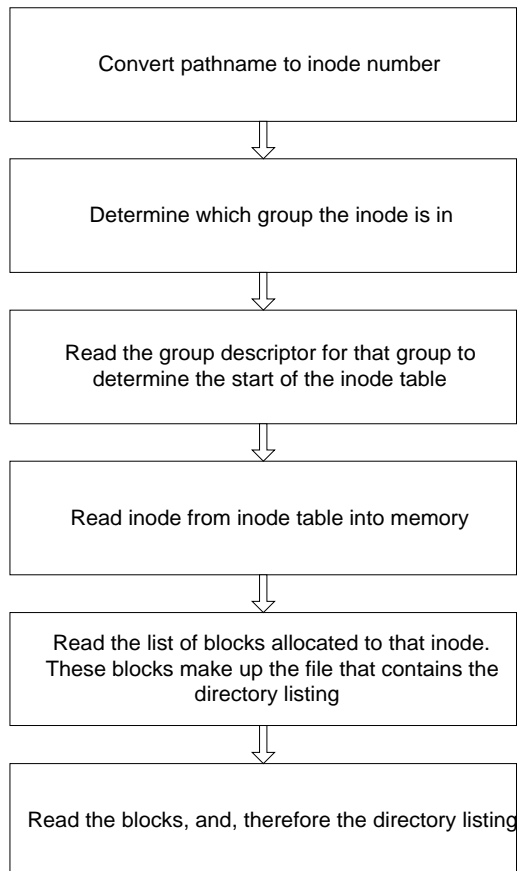
The foundation for this layer is the function to read a block from the filesystem. This interfaces layer 1 with layer 0. Under ext2fs, the block size is set when the filesystem is created, so it can take a number of different values. The function to read a block has to read a specified block from the filesystem, taking into account the block size. This function, therefore, performs one or more calls to the read logical sector function of layer 0 when called to read a block. As the read block function is used every time any operation needs to read a block from disk, it is vital that it is efficient. To improve performance a caching system is implemented within this function. For further details regarding the implementation of the cache please see section 5.11.

The first task that has to be performed by this layer is obtaining the superblock information. It is based upon the information that is contained in the superblock that the validity of the filesystem can be determined, also information contained in the superblock is needed to perform calculations necessary to even the most basic of filesystem operations. The superblock and the group descriptors are both read and buffered in memory when ext2lib is first loaded, for performance reasons.

To illustrate how the rest of the functions fit together to provide the filesystem commands of layer 2, and in so doing illustrate what is necessary of these functions and what they require, a mini case study will be used. That of

retrieving a directory listing. It should be noted that because directories are implemented as a special type of file in the ext2fs, this process is practically identical to that for reading a file.

#### 4.3.1 A Mini-case study: Obtaining a Directory Listing



The steps in the diagram on the right show the tasks necessary to read a directory.

Firstly, the inode number of the inode that represents the directory has to be found. For further details on how this is done, please refer to section 5.17. It is then necessary to read this inode from the inode table. However, to be able to do this, the group that the inode is in has to be determined. This is a simple calculation based on information held in the superblock, which is conveniently buffered in memory. Once the group has been determined, its group descriptor has to be read. These are also buffered in memory, but a function is necessary to locate and return the correct descriptor.

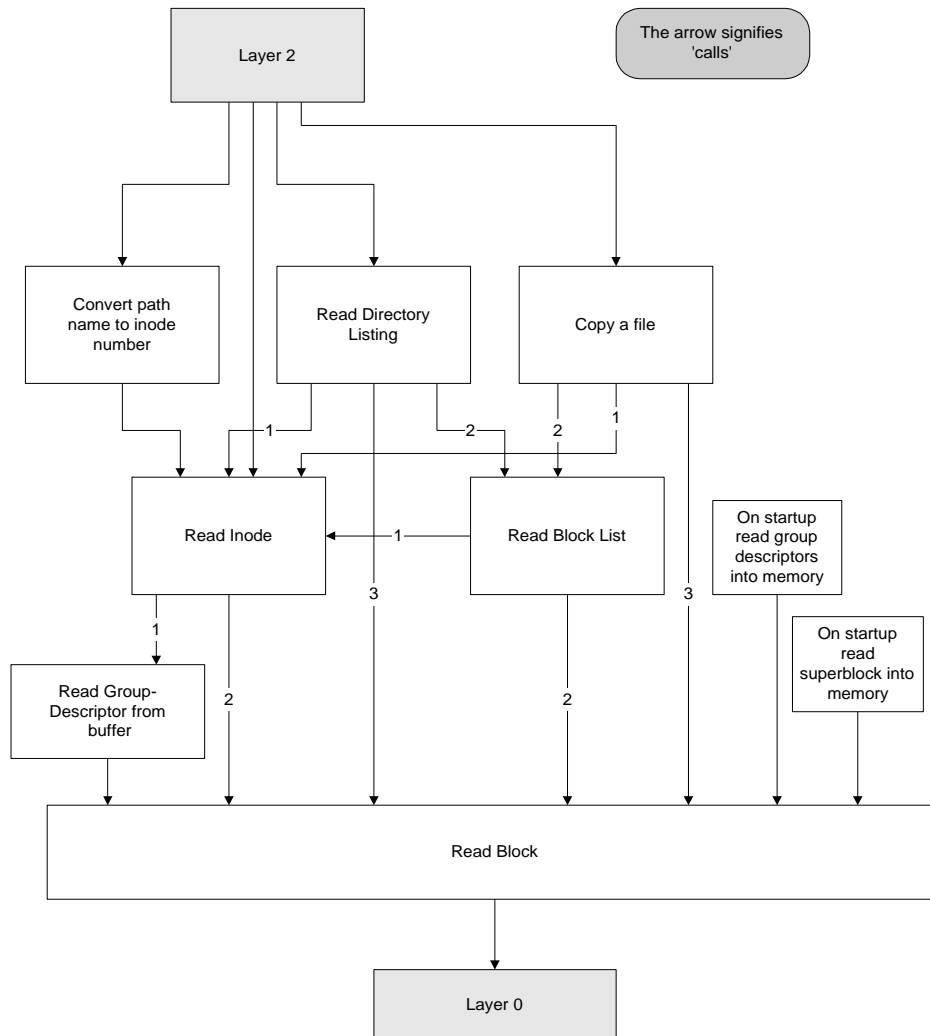
The group descriptor is necessary to determine the block address of the start of the inode table. Once this has been read, the required inode can be found and copied into memory.

Next, the list of blocks that the inode has allocated has to be read, so this means reading direct, indirect, and possibly double and triple indirect blocks (although this is unlikely in the case of a directory). So a function is necessary to read the complete list of blocks, and it must be designed to handle all the levels of indirection

Once this block list has been read, the actual blocks can be accessed. This is, in effect, reading the directory.

This mini case study has illustrated the need for the functions listed at the beginning of this section. The function of copying a file only differs in that on reading the blocks allocated to the inode, they are copied into an open file within the DOS filesystem.

The diagram below illustrates how all the functions shown to be required in the mini case study above are fitted together. The arrows in the diagram indicate when a function is calling another function. For functions that call more than one other function, the numbers on the arrows indicate the order in which the other functions are called. For example, for the case of copying a file, the function first reads the inode to determine file size, etc., then to copy the file it requires the list of allocated blocks for the inode so it calls the read block list function. Finally the blocks need to be read, and the copy function calls the read block function.



In addition to these functions that perform the operations on the ext2 filesystem, it is necessary to provide a path for layer 2 to obtain the partition information. Layer 2 cannot simply call the read partition information function from layer 0 as this compromises the integrity of the three layer approach. An additional function is supplied in layer 1 to retrieve the partition information and pass it to layer 2.

## 4.4 Layer 2 - The Application Interface Layer

This layer is designed to provide an interface for application programs to navigate the filesystem as if at a command line. It was decided to implement the application interface in this way because this removes the need for the calling application program (or programmer) to have to consider the internal workings of the ext2 filesystem. It simply provides a well known set of commands that can be used to navigate the filesystem. In order to achieve this layer 2 has to implement the following functions:

- A 'part' command to display partition information for all the hard disks connected to the PC.
- A 'mount' command to mount an ext2 filesystem
- An 'ls' command that returns a directory listing.
- A 'cd' command to change the current directory
- A 'cp' command to copy a file from a location on the ext2 filesystem to one on the DOS filesystem.
- A 'get inode' command to return the inode information for a particular file or directory

Each of the above commands is to function in a similar way to those found on the command line. The library keeps a record of the disk drive and partition numbers of currently mounted filesystem and also the current directory.

All the commands return information to the calling application, this is achieved by writing the information to a location in memory and passing a pointer to that location to the calling application.

### 4.4.1 The part command

This command has one simple function. It displays the partition information for all hard disks connected to the PC. That is, it returns the partition number, type and size. These are then used by the calling application to prompt the user as to which partition is to be mounted. This function simply calls a function supplied in layer 1, which, in turn, reads the partition information that was loaded into the memory buffer by layer 0. This information is then passed back to the calling application.

### 4.4.2 The mount command

This command is used by the calling application to mount an ext2 filesystem. This function calls the layer 1 functions to read the superblock into memory and to read the group descriptors into memory. It performs checks on the validity of the filesystem and returns information to the calling application

regarding the filesystem that has just been mounted. This information includes the filesystem size, the date of the last mount and the flag that indicates whether the filesystem contains errors.

#### 4.4.3 The ls command

This function acts in a similar way to the command line 'ls' or 'dir' commands. Its task is to return a directory listing to the calling application. A path can be supplied to this function and a listing of that directory will be returned, or if none is supplied, a listing of the current directory is returned.

To obtain a directory listing this function first calls the function in layer 1 to convert the path of the directory to an inode number, it then calls the read directory function to read the directory represented by that inode.

The directory listing is passed to the calling application.

#### 4.4.4 The cd command

This function changes the current directory to the one specified by the calling application. It performs a check to determine whether the specified path points to a valid directory, and if so, changes the stored current directory to the one specified.

To verify that the directory is a valid one, the layer 1 function is called to obtain the inode number of the inode representing the specified directory. Once the inode has been obtained, its type is one of the values it holds, so it can be easily determined if the inode represents a directory. To read the directory the read directory function in layer 1 is called to read the directory corresponding to the inode number specified.

The cd command returns the path of the current directory to the calling application.

#### 4.4.5 The cp command

This function copies a file from the ext2fs path specified to the DOS path specified. It first performs checks on the validity of both paths, and checks to see that the destination file does not already exist. If all is well, the file is copied from the source to the destination. If not an error code is returned to the calling application.

To copy the file the source path is converted to an inode number by the layer 1 function provided to do this. Then the copy file function in layer 1 is called

to copy the file with this inode number to the destination path on the DOS filesystem.

If the destination file exists an error is returned to the calling application, but an overwrite can be forced by changing the value of a flag passed to the cp function.

#### 4.4.6 The get inode command

This command returns the inode of the specified file or directory to the calling application. It simply calls the get inode function of layer 1 and passes the inode to the calling application. This function is included as this gives the calling application a method of retrieving detailed information about a specific file or directory.

## 5. Ext2lib Implementation

This section gives a detailed description of how the features outlined in the design section are implemented in the ext2lib library. The aim of this section is not to give a line-by-line account of the code, but to highlight important sections and describe these in detail.

### 5.1 The Structure of the Source Code

The source code is structured by layer and by task. The source file names are prefixed with either L0, L1 or L2 representing source for layer 0, 1 and 2 respectively. The rest of the filename indicates the task carried out by the functions within that file. Each c source file (.c) has an accompanying header (.h) file. A brief description of each is shown below:

- L0-dskio.c - Low level disk access routines
- L0-part.c - Routines for handling the partition tables
- L0-logic.c - The logical sector reading routine
- L1-sblk.c - Routines to handle the superblock
- L1-gpdes.c - Routines for handling the group descriptors
- L1-inode.c - Routines for reading and handling inodes
- L1-dir.c - Routines to read a directory listing and handle path to inode conversion
- L1-copy.c - Routine to copy a file from the ext2 filesystem to the DOS filesystem
- L1-funcs.c - Routines used by numerous L1 functions, including the ReadBlock function
- L2-mount.c - The 'mount' and 'part' command routines
- L2-cd.c - The 'cd' command routine.
- L2-ls.c - The 'ls' command routine.
- L2-cp.c - The 'cp' command routine.
- Libmain.c - The libmain initialisation function.
- Layer\_0.h - The data types, prototypes and definitions that are exported from layer 0 to layer 1
- Layer\_1.h - The data types, prototypes and definitions that are exported from layer 1 to layer 2
- Ext2lib.h - The data types, prototypes and definitions that are exported to the calling application from the ext2lib library
- Ext2\_fs.h - The definitions and data structures for the ext2 filesystem.

Some of the types and macro definitions from windows.h are used.

None of the header files from a given layer are allowed to be included by source files from another layer to maintain the integrity of the three layer

structure. Instead, a header file is supplied for each of layers 0 and 1 (layer\_0.h and layer\_1.h, respectively) that can be included in the source files of the next layer up. So, layer\_0.h is included by layer 1 functions, and layer\_1.h is included by layer 2 functions. These header files define the data structures and types and the function prototypes that are allowed to be used by the next layer.

A header file, ext2lib.h, is also supplied. This is not included by any of the source for the library, but is provided for inclusion in the source for the calling application. It prototypes the exported functions of ext2lib and defines the data types that these functions return to the calling application.

The data types and definitions used by the ext2 filesystem are defined in a source file called ext2\_fs.h. This is part of the linux kernel source, and was written by Remy Card. It is included by most of the source files for ext2lib, and also must be included in the source for the calling application wherever ext2lib.h is included. The copyright notice from the beginning of the file is shown here:

```
/*
 * linux/include/linux/ext2_fs.h
 *
 * Copyright (C) 1992, 1993, 1994 Remy Card
 (card@masi.ibp.fr)
 * Laboratoire MASI - Institut Blaise Pascal
 * Universite Pierre et Marie Curie (Paris VI)
 *
 * from
 *
 * linux/include/linux/minix_fs.h
 *
 * Copyright (C) 1991, 1992 Linus Torvalds
 */
```

## **5.2 The implementation of layer 0: The Hardware access layer**

The first function to be implemented was the function to read a physical sector from the disk. The whole application is underpinned by this function as it is used whenever data has to be read from the disk. The function has to perform low level accesses on non-DOS hard disks, so this has to be performed at the BIOS level. This is quite simple under MS-DOS, but under Windows 95 presents a few problems.

## **5.3 Non-DOS Hard Disk Access Under MS-DOS: BIOS Disk Functions**

The principles behind talking to non-DOS disks under windows 95 is very similar to the method used under MS-DOS. MS-DOS does not assign drive letters, or recognise in any other way the existence of non-DOS disks connected to the PC. This means that Linux partitions cannot be seen at all, therefore it is

necessary work at a level lower than the DOS to be able to perform disk I/O on Linux partitions.

The basic method consists of calling Interrupt 0x13 to access the BIOS (Basic I/O Subsystem) disk routines. This interrupt provides access to many disk functions (see DOS interrupt list for more details), but the main point of interest here is the function to read a sector from disk. To activate an interrupt, certain registers have to be loaded with the values necessary to perform the operation. To perform a disk read the following registers have to be loaded with these values:

AH = 0x02  
AL = number of sectors to read (must be nonzero)  
CH = low eight bits of cylinder number  
CL = sector number 1-63 (bits 0-5)  
    high two bits of cylinder (bits 6-7, hard disk only)  
DH = head number  
DL = drive number (bit 7 set for hard disk)  
ES:BX -> data buffer

Reference: [1]

The AH register specifies the function to be performed, it is loaded with 0x02 as this is the value that corresponds to the interrupt 0x13 read sector function. The drive number starts at 0x80 for the first hard disk, 0x81 for the second, etc. ES:BX are loaded with the memory location to store the data read from disk.

Provided that valid values are loaded into the registers, after issuing interrupt 0x13 the data read from the disk is stored in the location in memory pointed to by ES:BX and can be further manipulated as necessary.

#### **5.4 Using Interrupt 0x13 Under MS Windows 95: DPMI**

Windows 95, like previous versions of Windows, does not support calling BIOS disk functions to gain access to hard disks from Win16 and Win32 applications. The reason is that BIOSXLAT.VXD does not translate BIOS requests to hard disks from protected-mode into V86 mode, and this causes the ROM BIOS to be called with an invalid address. This means that the method of issuing an interrupt 0x13 call directly as described above does not work.

However, help is at hand in the form of DPMI - The DOS Protected Mode Interface. This is available to win16 executables or libraries running under Windows 3.x and Windows 95. DPMI has a function called 'simulate real-mode interrupt' and this can be used to call real-mode BIOS disk functions using interrupt 0x13. When DPMI is used to call interrupt 0x13 BIOS disk functions, BIOSXLAT.VXD is bypassed and the real-mode BIOS is called, thus solving the translation problem.

DPMI is accessed through another interrupt - interrupt 0x31. Again registers have to be loaded with the correct values before executing the interrupt. These values are:

AX = 0x0300

BL = interrupt number

BH = flags

bit 0: reset the interrupt controller and A20 line (DPMI 0.9)  
reserved, must be 0 (DPMI 1.0+)

others: reserved, must be 0

CX = number of words to copy from protected mode to real mode stack

ES:(E)DI = selector:offset of real mode call structure

Reference: [1]

The values for the real-mode (interrupt 0x13) interrupt are placed in a structure called the real-mode call structure. This structure contains elements corresponding the registers. The values are assigned to these variables that would normally be loaded into the registers when performing an interrupt 0x13 call. Using this method DPMI allows access to the hard disk as if a genuine interrupt 0x13 was being called. Using DPMI to simulate interrupt 0x13 calls it can be seen that it is possible for win16 libraries or executables to directly access non-DOS hard disks from within windows. It is upon this method that the low level disk access sections of the ext2lib library were built.

### 5.5 Layer 0: The *SimulateRMInt* Function

It is this function that implements the simulate real mode interrupt function of DPMI. This function will simulate any real mode interrupt, but is used exclusively to implement interrupt 0x13 calls to access the BIOS disk functions.

It is called from the ReadPhysicalSector function to actually call the BIOS to perform the disk read. To simulate the interrupt 0x13 call, the DPMI simulate real mode interrupt function requires a data structure called the Real Mode Call Structure (RMCS) to be filled with the register values for the real mode call. This data structure is shown below:

```
typedef struct tagRMCS {  
    DWORD edi, esi, ebp, RESERVED, ebx, edx, ecx, eax;  
    WORD  wFlags, es, ds, fs, gs, ip, cs, sp, ss;  
} RMCS, FAR* LPRMCS;
```

As can be seen from the above structure, it has variables that correspond to the cpu registers that would normally be loaded with values prior to issuing an interrupt call. This structure is then used to make the interrupt 0x31 call to

perform the simulate real mode interrupt function. The function is passed two values by the calling function: the number of the interrupt to simulate, `bIntNum` and a pointer to the Real Mode Call Structure, `lpCallStruct`. These are then used by a small section of assembly code to perform the interrupt call.

```
_asm {
    push di
    mov ax, 0x300           // DPMI Simulate Real Mode Int
    mov bl, bIntNum        // Number of the interrupt to simulate
    mov bh, 0x01          // Bit 0 = 1; all other bits must be 0
    xor cx, cx             // No words to copy
    les di, lpCallStruct
    int 0x31              // Call DPMI
    jc  END1              // CF set if error occurred
    mov fRetVal, TRUE
END1:
    pop di
}
```

This performs the task of loading the appropriate registers with the values described in the above section on 'Using Interrupt 0x13 Under MS Windows 95: DPMI', and then issuing the interrupt 0x31 call. As can be seen from the above code, the registers are simply loaded with the required values for the function number for the simulate real mode interrupt function, the number of the interrupt to simulate and a pointer to the RMCS. If the interrupt produces an error, the carry flag (CF) is set.

## 5.6 Layer 0: The *ReadPhysicalSector* Function

This function is called by the `ReadLogicalSector` function to read a physical sector from disk. It takes six arguments:

- `bDrive` - The drive number, 0x80 for 1<sup>st</sup> HDD, 0x81 for 2<sup>nd</sup>, etc..
- `wCyl`, `wHead`, `wSec` - The CHS reference for the sector to be read.
- `lpBuffer` - The pointer to the buffer to read the sector data into.
- `cbBuffSize` - The size of the buffer.

After validating the disk parameters passed to the function, the next task is to load the Real Mode Call Structure (RMCS) with the appropriate values to make the interrupt 0x13 call. The values that must be loaded are listed in the section 'Using Interrupt 0x13 under MS-DOS' above, but are listed again here for clarity:

AH = 0x02

AL = number of sectors to read (must be nonzero)

CH = low eight bits of cylinder number  
CL = sector number 1-63 (bits 0-5)  
    high two bits of cylinder (bits 6-7, hard disk only)  
DH = head number  
DL = drive number (bit 7 set for hard disk)  
ES:BX -> data buffer

The values are loaded by the section of coe shown below:

```
callStruct.eax = 0x0201;           // BIOS read, 1 sector
//ch = low 8 bits of cyl # cl = hi 2 bits of cyl # and sect num
callStruct.ecx = MAKEWORD(((wCyl & 768) >> 2) | wSec), (wCyl & 255));
callStruct.edx = MAKEWORD(bDrive, wHead); // Head #, Drive #
callStruct.ebx = LOWORD(RMlpBuffer); // Offset of sector buffer
callStruct.es  = HIWORD(RMlpBuffer); // Segment of sector buffer
```

On studying the above code, it is clear that the ax, cx and dx registers are loaded with the values listed above. To calculate the value to load into the cx register, two bitmasks are used, 768 which corresponds to binary 1100000000 and 255 which corresponds to binary 0011111111. These are used to mask off the appropriate bits of the cylinder number in order to calculate the value that corresponds to the low 8 bits of the cylinder number for the high byte of cx, and the combination of the sector number and hi 2 bits of the cylinder number for the low byte of cx. The bx and es registers are loaded with the segment and offset of the pointer to the data buffer. Note that this pointer has to be a real-mode pointer for use by the BIOS, rather than the protected-mode pointer used by the library code.

The remainder of the function simply calls SimulateRMInt to issue the interrupt.

### 5.7 Layer 0: The GetDiskParams Function

The GetDiskParams function calls the BIOS disk function to retrieve the physical disk geometry for each drive connected to the PC. The values recorded are the maximum cylinder, head and sector numbers. These are read into a global array of data structures called DISKINFO, the definition of this type is shown below:

```
typedef struct diskinfo
{
    DWORD rgDskParams[4];
    struct firstsector FirstSec;
} DISKINFO;
```

A DISKINFO structure is used for each disk on the system and they are held in an array of MAX\_DISKS length. The disk geometry is stored in the array rgDiskParams, The values are as follows:

- rgDiskParams[0] = drive number
- rgDiskParams[1] = maximum cylinder number
- rgDiskParams[2] = maximum head number
- rgDiskParams[3] = maximum sector number

The FirstSec value is discussed in section 5.8

The BIOS disk function to retrieve the disk parameters is function 0x08 of interrupt 0x13. The RMCS is loaded with the following values

AH = 0x08  
DL = drive (bit 7 set for hard disk)

Reference: [1]

SimulateRMInt is then called to issue the interrupt. The following values are returned by the interrupt:

CF set on error  
AH = status (0x07)  
CF clear if successful  
AH = 00h  
AL = 00h on at least some BIOSes  
BL = drive type (AT/PS2 floppies only)  
CH = low eight bits of maximum cylinder number  
CL = maximum sector number (bits 5-0)  
          high two bits of maximum cylinder number (bits 7-6)  
DH = maximum head number  
DL = number of drives

Reference: [1]

These are copied into the rgDiskParams array This process is repeated for each disk found by the BIOS. The function then returns TRUE for success or FALSE if the SimulateRMInt call was unsuccessful.

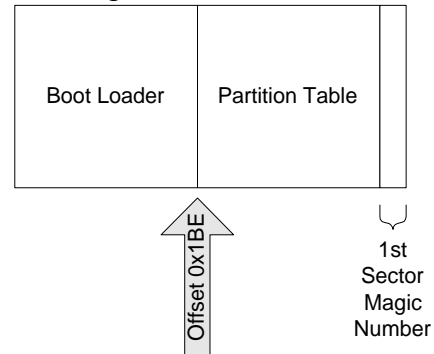
GetDiskParams is called once during execution of the library, when the library is first loaded.

### 5.8 Layer 0: The ReadPartInfo Function

This function uses the ReadPhysicalSector function to read the first sector from each disk recorded in the rgDiskParams array into the firstsector structure. The DISKINFO data type, introduced in the above section, is used to store the drive geometry and the first sector of each hard disk connected to the PC. The firstsector structure stores this first sector. Its structure is shown below:

```
struct firstsector {
    unsigned char Fill[0x1be];
    struct partition Part[4];
    unsigned short Magic;          /* Must be FIRST_SECT_MAGIC */
};
```

The first sector of a hard disk contains three things, the boot loader, used by the BIOS to boot the operating system from the first hard disk, the partition table for that disk and the first sector magic number. The firstsec structure is designed to represent this. By reading the first sector from disk straight into the memory location of the firstsec structure, the values for the partition table and the magic number are loaded into the correct places. A hard disk has space for four entries in its partition table, and so an array of four partition structures is used to hold the partition table. The partition structure is organised in such a way that the values all fall into the right places when the sector is read, this structure is shown below:



```

struct partition {
    unsigned char boot_ind;        /* 0x80 - active */
    unsigned char head;          /* starting head */
    unsigned char sector;        /* starting sector */
    unsigned char cyl;          /* starting cylinder */
    unsigned char sys_ind;        /* What partition type */
    unsigned char end_head;      /* end head */
    unsigned char end_sector;    /* end sector */
    unsigned char end_cyl;      /* end cylinder */
    unsigned short start_sectlo; /* starting sector counting from 0 */
    unsigned short start_secthi; /* starting sector counting from 0 */
    unsigned short nr_sectsl;   /* nr of sectors in partition */
    unsigned short nr_sectsh;   /* nr of sectors in partition */
}

```

The values are as indicated by the comments. Notice that the start sector and number of sectors values are stored in the typical lo byte - hi byte form, as they are on disk. This structure is then used by the GetPartTable function to return a more convenient form for the partition table.

The 1<sup>st</sup> sector magic number, stored in the variable Magic, is used to check the validity of the first sector, it's value is 0xAA55. See section 2 for more information on the structure of hard disks.

This function is called only once, when the library is first loaded.

## 5.9 Layer 0: The *GetPartTable* Function

This function is used to retrieve the partition table for a disk. It reads the information stored in the DiskInfo array and returns a much more convenient form for the partition table on the disk. The data structure used for this is given the type PARTTBL, it represents one entry in the partition table for the disk. An array of four such structures is used to record the entire partition table. The structure is shown below:

```
typedef struct part_table {
    unsigned uBoot;
    unsigned uStartHead;
    unsigned uStartSec;
    unsigned uStartCyl;
    unsigned uPartType;
    char      szPartType[11];
    unsigned uEndHead;
    unsigned uEndSec;
    unsigned uEndCyl;
    unsigned long ulStartSec;
    unsigned long ulNumSec;
}PARTTBL;
```

The meanings of the variables is as follows:

- uBoot - The bootable status of the partition.
- uStartHead, uStartSec, uStartCyl - The CHS reference for the start of the partition
- uPartType - The hex value for the partition type, see section 2.2
- szPartType - A string indicating the partition type, this takes one of the following values:
  - DOS - For a primary DOS partition
  - DOSEXT - For an Extended DOS partition
  - LINUX - For a Linux filesystem partition
  - LINUXSWP - For a Linux swapfile partition
- uENdHead, uENdSec, uENdCyl - The CHS reference for the end of the partition.
- ulStartSec - The start sector of the partition, sectors numbered from 0.
- ulEndSec - The end sector of the partition, sectors numbered from 0.

The function accepts the PartTable array and the drive number as arguments and simply reads the partition table values from the DiskInfo array and calculates the ulStartSec and ulEndSec values, then fills the PartTable array and returns.

## 5.10 Layer 0: The ReadLogicalSector Function

The ReadLogicalSector function provides the interface between layer 0 and layer 1. It reads a logical sector from a specified drive number and partition number. It accepts five arguments:

- bDrive - The Drive number, 0x80 for 1<sup>st</sup> HDD, 0x81 for 2<sup>nd</sup>, etc.
- bPart - The partition number, 0 for 1<sup>st</sup> partition, 1 for the 2<sup>nd</sup>, etc.
- chSecNum - The sector number of the sector to be read.
- lpBuff - The buffer into which to read the sector data.
- chBuffSize - The size of the buffer

This function must calculate the CHS reference that corresponds to the logical sector to be read, then calls ReadPhysicalSector to perform the disk read.

First the absolute sector number (sectors numbered from 0 from the 1<sup>st</sup> sector on the disk) is calculated. This is simply done by adding the ulStartSec value for the specified partition to the logical sector number, ulSecNum. This absolute sector number must then be converted to a CHS reference for use by the ReadLogicalSector function. This is done using the calculation shown below:

```
//perform sector # to CHS translation
ulPhSec = chSecNum % DiskInfo[iDrive].rgDskParams[3] + 1;
chSecNum /= DiskInfo[iDrive].rgDskParams[3];
ulPhHead = chSecNum % (DiskInfo[iDrive].rgDskParams[2] + 1);
ulPhCyl = chSecNum / (DiskInfo[iDrive].rgDskParams[2] + 1);
```

For more information on how the absolute sector number translates to a CHS reference, please see section 2. For more information on the DiskInfo array, please see section 5.7

Once the CHS reference has been calculated, the ReadPhysicalSector function is called to perform the disk read.

## 5.11 Layer 1: The ReadBlock Function

This function reads a logical block from the mounted filesystem. It must take into account the block size, which can vary from filesystem to filesystem. Typical values for the block size are 1024, 2048 or 4096 bytes. This function provides the interface between layer 0 and layer 1 and is used by any operation in layer 2 that needs to access the filesystem on disk. It calls the layer 0 function ReadLogicalSector to perform the disk reads, one or more of which may be necessary to read a complete block. To read the correct sectors from disk, two values need to be calculated. The start sector (ulSector) and the number of sectors to read (uNumSec). These are calculated as shown below:

```
ulSector = ulBlkSize * ulBlock / SECTOR_SIZE;
uNumSec = ulBlkSize / SECTOR_SIZE;
```

Where `ulBlkSize` is the block size, `ulBlock` is the block address of the requested block.

The main bottleneck to the performance of the `ext2lib` library is in reading from disk. This is a slow operation as the hard disk is many times slower than accessing memory. To minimise the number of disk reads that must be performed, the `readblock` function implements two caches of blocks.

#### 5.11.1 The Read-Ahead Cache

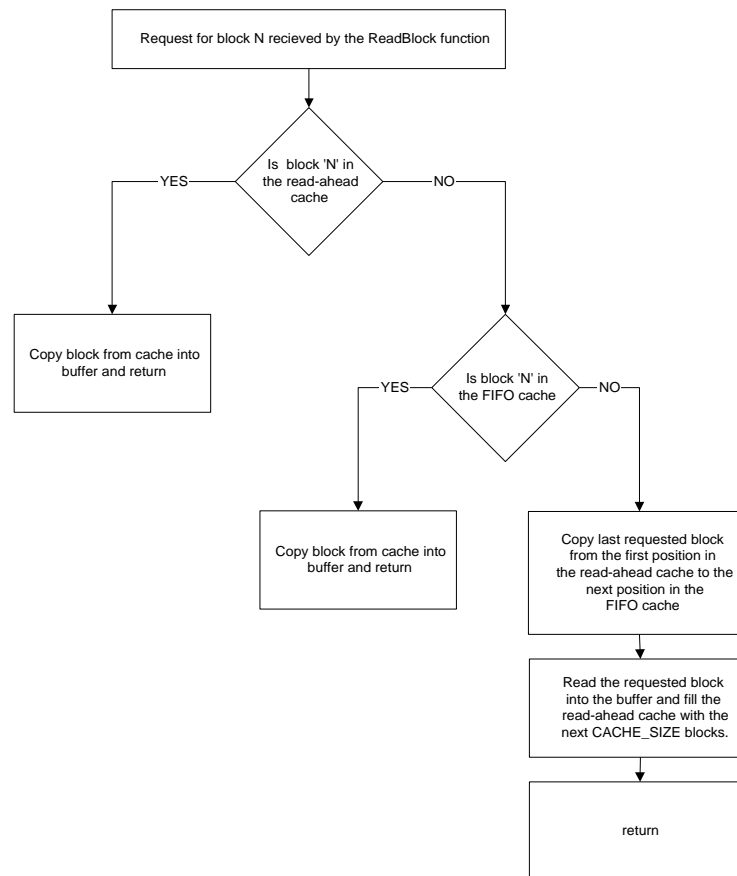
It was observed during testing that many disk accesses require a number of contiguous blocks to be read from the disk, especially when reading directories or files from the `ext2` filesystem. It is for this reason that a read-ahead cache was implemented. When a request to read a block from disk is first received by the `ReadBlock` function, it not only reads the block requested, but also reads `CACHE_SIZE` blocks (currently 8 blocks) ahead and buffers these blocks in memory. This does not really take much more time than reading one block, as the blocks being read are contiguous, the number of seeks that the disk head has to make is minimised. An index to the cache holds the block addresses of the cached blocks in an array.

The next time a block is requested, the `ReadBlock` function checks the cache index to determine if the requested block is cached, and if it is, simply reads it from memory, rather than accessing the disk again, which provides a great boost to performance. If the block is not cached, the block is read from disk and the read-ahead cache is re-filled.

#### 5.11.2 The FIFO Cache

Whilst navigating the filesystem using the `cd` command and `ls` command to read directories, blocks that were requested maybe two or three requests ago are again requested from the `ReadBlock` function. It would provide a boost to performance if these blocks were cached, as they would not have to be re-read from disk. It is for this reason that the FIFO cache was implemented. It buffers the last `CACHE_SIZE` (currently 8) requested blocks in memory. Each time a new block is requested, the index for the FIFO cache is checked, and if the block has been cached it is simply copied from the cache. If a block is not in the FIFO cache, it is read from disk and the last block requested (the first block in the read-ahead cache) is stored in the next position in the FIFO cache. If the cache is full, the next cached block is written at the beginning of the cache.

The diagram below illustrates how these two caches operate in the `readblock` function:



These caches provide a good boost to the performance of the ext2lib library, as they greatly reduce the amount of disk reads necessary.

### 5.12 Layer 1: The ReadSuperBlock and GetSuperBlock Functions

The ReadSuperBlock function is called once, when a filesystem is mounted. It reads the superblock information from block 1 of the filesystem into memory. The data structure used to hold the superblock is given a type, SUPERBLOCK. The form of this structure is defined in ext2\_fs.h, and is stored in memory in the same form as it is stored on disk. The superblock is stored in a variable, SuperBlock, of this type that is global to all the layer 1 functions.

The GetSuperBlock function is used by layer 2 functions to read the superblock information. The layer 2 functions cannot simply read it from the SuperBlock variable, as this is only global to layer 1 to preserve the integrity of the three layer structure. It simply copies the superblock information from the variable SuperBlock into the memory location pointed to by lpSBlock, which is passed to the function.

Please see section 3.3.2 for details of the superblock structure

### 5.13 Layer 1: The ReadGroupDesc and GetGroupDesc Functions

The ReadGroupDesc function simply reads the filesystem group descriptors from disk into a memory buffer. It is called once when the filesystem is mounted.

The GetGroupDesc function is used to retrieve a specified group descriptor from the memory buffer filled by ReadGroupDesc. It simply calculates the offset within the buffer of the required group descriptor, identified by the variable uGroup, and copies the descriptor from the buffer to the memory location pointed to by GroupDesc. The group descriptor structure, defined in ext2\_fs.h is assigned a type, GROUPDESC.

Please see section 3.3.3 for details of the group descriptor structure.

### 5.14 Layer 1: The GetInode Function

This function is used to read a specified inode from the disk into the Inode structure passed to the function. The inode is specified by passing its inode number to the function. The inode structure, defined in ext2\_fs.h, is given a type: INODE. For more details on the structure of the inode, see section 3.3.5.

The first task that this function has to perform is that of calculating which block group the requested inode is in. This is achieved by dividing the inode number by the number of inodes in each group. The integer part of this result gives the group in which the inode is located. The group descriptor for this group is then read by calling the GetGroupDesc function.

Next, the block address of the block containing the inode has to be calculated so that the disk read can be performed. This is done by the code shown below

```
//Calculate block number
ulBlkSize = 1024 << SuperBlock.s_log_block_size;
ulInodesPerBlk = ulBlkSize / sizeof(INODE);
ulInodeOffset = ulInodeNum - (ulGroup * SuperBlock.s_inodes_per_group);
ulBlock = GroupDesc.bg_inode_table + floor((double)ulInodeOffset /
                                           (double)ulInodesPerBlk);
```

Firstly the number of inodes per block, ulInodesPerBlk, is calculated. Then, as each group only contains a portion of the inode table, the offset within the inode table for this group, ulInodeOffset, has to be calculated. This offset is measured in inodes. To calculate it, the total number of inodes contained within groups previous to this one is determined and then subtracted from the inode number of the requested inode. Finally, to determine the block address of the block containing the inode, the group descriptor value, bg\_inode\_table is used. This gives the block address of the beginning of the portion of the inode table

contained within this group. To calculate the exact block required, the inode offset is divided by the number of inodes per block, this gives a block offset within the inode table. This result is added to the `bg_inode_table` value to produce the final block address of the block containing the required inode.

The block containing the inode can now be read into a memory buffer using the `ReadBlock` function. To aid performance the `GetInode` function caches the last block that was read by the previous call to the function. If the next inode is from the same block, no disk read is necessary as the block is already cached in memory and can be reused.

The final task for this function to perform is copying the specified inode from the block buffer to the `Inode` variable. The byte offset off the required inode within the buffered block has to be calculated, however. This is achieved as shown below:

```
//calculate offset of desired inode within buffered block
ulBlkOffset = sizeof(INODE) * (ulInodeOffset - 1 -
    ((ulBlock - GroupDesc.bg_inode_table) * ulInodesPerBlk));
```

The first result that is required is the number of inodes that are in the buffered block before the required inode. This is calculated by first determining the number of blocks in the inode table that lie before the buffered block. This is achieved by subtracting the block address of the beginning of the inode table, `bg_inode_table`, from the block address of the buffered block, `ulBlock`. This result is then multiplied by the number of inodes per block, `ullNodesPerBlk`, to give the number of inodes that lie in the blocks before the buffered block. This result is then subtracted from the inode offset, `ullNodeOffset`, minus 1. This gives the number of inodes that are in the buffered block before the required inode, the result that is required. This is then simply multiplied by the size of an inode to give the byte offset.

This byte offset is then used to copy the desired inode from the block buffer into the `Inode` structure.

### **5.15 Layer 1: The `GetBlockList` Function**

This function is used to read the complete list of block addresses for the blocks that are allocated by a particular inode. These are the direct blocks, contained within the inode itself, and also the indirect, double indirect and triple indirect blocks. The function reads the list of block addresses into a memory buffer and returns a pointer to this buffer, `lpBlocksBuff`. The memory required for the block buffer is dynamically expanded according to how many block addresses are read into it and an offset, `ulBuffOffset`, (measured in unsigned

longs) is used to keep track of the next position in the memory buffer to read block addresses into. Each time a block is read into the buffer `ulBuffOffset` is incremented by `ulLongPerBlk`. This variable stores the number of unsigned longs that a block contains.

The addresses of the direct blocks are simply read from the first 12 elements of the 'blocks' array contained in the inode. (For details of the inode structure please see section 3.3.5).

The next element of the array (element 12) gives the block address of the indirect blocks. This is a block of block addresses of allocated blocks. See section 3.3.5 for more information on how the inode stores allocated block addresses. These block addresses are read into the memory buffer using the `ReadBlock` function.

Element 13 of the array gives the block address of the block of double indirect blocks. This block contains pointers to blocks containing the block addresses of the actual allocated blocks. This block is read and buffered in a separate memory buffer, `lpBuff`. Each block of direct blocks that the elements in `lpBuff` hold the block addresses of is then read into the `lpBlocksBuff` memory buffer. This operation is performed by the code section below:

```
//buffer up block of indirect blocks
if(!ReadBlock(bDrive, bPart, Inode.i_block[13], lpBuff))
    {farfree(lpBuff);
    return NULL;}

while((lpBuff[iBlk] != 0) && (iBlk <= ulLongPerBlk))
    {
    //expand memory buffer
    lpTemp = farrealloc(lpBlocksBuff, (ulBuffOffset * uLongSize) + ulBlkSize);
    if (lpTemp != NULL)
        lpBlocksBuff = lpTemp;
    else
        {farfree(lpBuff);
        return NULL;}

    //read direct blocks
    ReadBlock(bDrive, bPart, lpBuff[iBlk], lpBlocksBuff + ulBuffOffset);
    ++iBlk;
    ulBuffOffset += ulLongPerBlk;
    }
}
```

Element 14 of the array gives the block address of the block of triple indirect blocks. This works in a similar way to the double indirect blocks described above, except with another layer of indirection. This block is buffered in a buffer pointed to by `lpTripBuff`. Each element of this buffer points to a block of double indirect blocks. Each of these blocks of double indirect blocks is processed as described for double indirect blocks, above.

Once the complete list of block addresses has been read into the memory buffer pointed to by `lpBlocksBuff`, this pointer is returned to the calling function.

## 5.16 Layer 1: The ReadDir Function

This purpose of this function is to read the directory listing pointed to by a specified inode. The inode number of the required directory is passed to the function. The function returns a small structure with the type DIRECTORY. This structure is defined as follows:

```
typedef struct directory {
    unsigned long ulDirLen;
    LPBYTE lpDirData;
} DIRECTORY;
```

It contains two elements, ulDirLen is the directory list length in bytes and lpDirData is a pointer to the memory location that the directory list is stored in.

The first operation that this function must carry out is to retrieve the inode that corresponds to the inode number, ullNodeNum. This is achieved by calling the GetInode function. Next the list of the blocks allocated to the inode is obtained by calling the GetBlockList function. These blocks are then read into the memory buffer, lpDirData, using a while loop to call the ReadBlock function for each block in the block list. The ulDirLen variable is copied from the i\_size field in the Inode variable.

The Directory structure is then returned to the calling function. This function makes up part of the interface between layers 1 and 2.

The structure of a directory is described in section 3.3.6, but a brief description is given here. Directories are implemented in ext2fs as a special kind of file. The structure of the file is a linked list of directory entries. This method is used rather than placing the directory entries one after another to save disk space, as ext2 filenames can be up to 255 characters long, but many are shorter. The structure of the directory entry is shown below:

```
/*
 * Structure of a directory entry
 */
#define EXT2_NAME_LEN 255

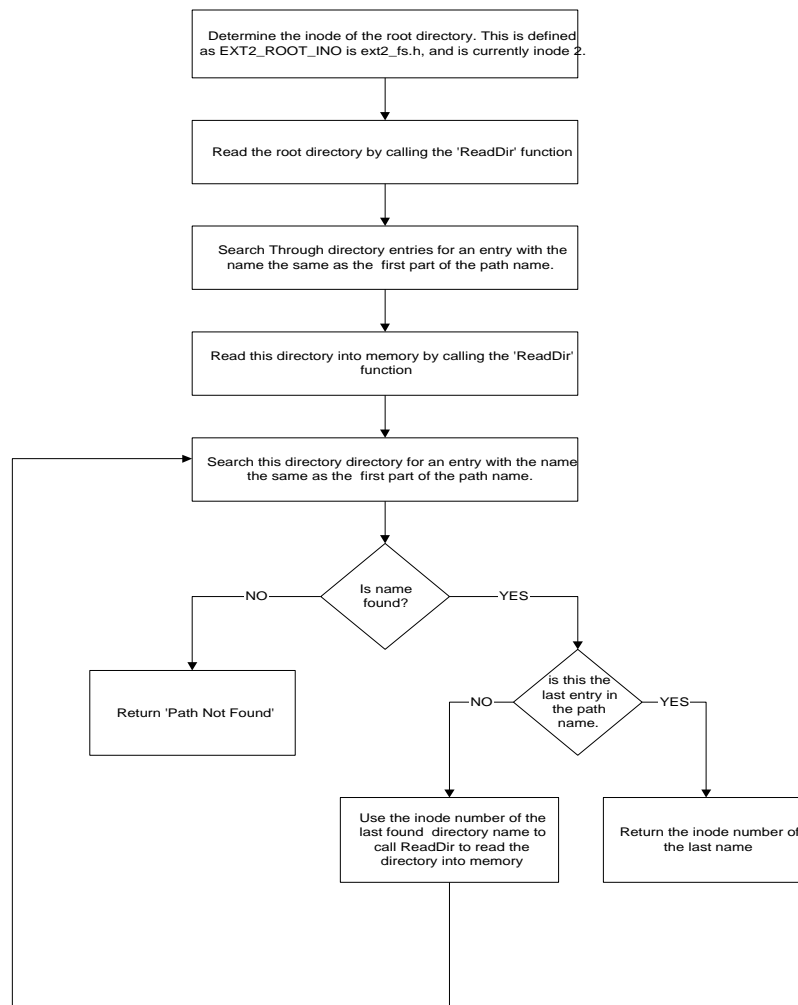
struct ext2_dir_entry {
    unsigned long inode;           /* Inode number */
    unsigned short rec_len;       /* Directory entry length
*/
    unsigned short name_len;      /* Name length */
    char name[EXT2_NAME_LEN];    /* File name */
};
```

The length of the directory name is stored in the variable name\_len. In order to navigate through successive directory entries, a variable to record the offset of the next entry is simply a cumulative total of these rec\_len values. The name\_len value is recorded because the character array holding its name is not

null terminated. Any function in a calling application that has to read the directory must use the `rec_len` values to read the directory.

### 5.17 Layer 1: The PathToInode Function

This is a very important function, as it performs the process of path to inode resolution. This is the process of locating the inode number of a file or directory given as a path. The steps for this process form part of the mini case study in section <SECTION>. The process is represented here in diagram format.



To perform this operation the path has to be parsed. This is done by searching through the path for the `ext2fs` directory separator, `'/'`, using the `_fmemccpy` function, and then recording an offset in the pathname for the beginning of a new file or directory name.

The resolution algorithm is implemented using a while loop to perform the directory entry scan nested within another while loop used to read successive parts of the filename. This works so:

```
while (next_filename_offset is less than path_length)
{
    read 'Nth' directory;
    read (N+1)th filename;
    while(directory_offset is less than directory_length)
    {
        read next directory entry;
        if( (N+1)th filename = current directory entry name)
        {
            next_inode = current_directory_entry's_inode;
            break;
        }
    }
    increment N;
}
return next_inode;
```

Looking at the source code for this function, it is clear to see where all the above pseudocode instructions have been implemented. The function returns the inode number of the inode representing the file or directory. This function makes up part of the interface between layers 1 and 2.

### 5.18 Layer 1: The CopyFile Function

This file is used to copy a file from the ext2 filesystem to a DOS filesystem. As arguments it takes the inode number of the source file, the DOS path (including drive letter) of the destination file and a flag to indicate whether the function is to overwrite the destination file if it already exists. The copy file function returns an error code, which is defined in L1-copy.h:

```
#define INODE_NOT_FOUND 1
#define BLK_LIST_ERR 2
#define OPEN_FILE_ERR 3
#define FILE_EXISTS 4
#define PATH_NOT_FOUND 5
```

The first two tasks that this function has to perform are retrieving the inode for the source file, and the block list for this inode. It does this with calls to GetInode and ReadBlockList, respectively.

The next operation is to test if the source file exists, this is done by attempting to open the source file for reading. If the overwrite flag, fOWrite, is

false the FILE\_EXIST error is returned. If the fOwrite flag is true, the function proceeds to copy the new file regardless of the destination file already existing, overwriting the destination file. The destination file is opened for writing using the standard C fopen function.

The blocks in the block list for the source file are then read using the ReadBlock function and written to the destination file using the fwrite function. A variable ulBytesLeft records the number of bytes left to copy, and is decremented by a block size every time a block is copied. If the number of bytes left to copy is less than a full block, the fwrite function only writes the exact number of bytes left using the fwrite command. This is to ensure that there is no garbage copied to the destination file from the slack space on the source file. This is implemented as shown in the code section shown below:

```
while((lpBlocksBuff[iBlk] > 0) && (ulOffset/ulBlkSize < Inode.i_blocks))
{
    ReadBlock(bDrive, bPart, lpBlocksBuff[iBlk], lpBuff);

    if(ulBytesLeft >= ulBlkSize)
        fwrite(lpBuff,ulBlkSize,1,pDest);
    else
        fwrite(lpBuff,ulBytesLeft,1,pDest);

    ulBytesLeft -= ulBlkSize;
    ++iBlk;
}
```

The function then closes the destination file and returns 0 if the file was copied correctly. This function forms part of the interface between layers 1 and 2.

### 5.19 Layer 1: The GetPartInfoNice Function

This function forms part of the interface between layers 1 and 2. It gives the layer 2 part command access to the partition table information. The part command is used to display the partition table information for all the disks connected to the PC. The layer 2 function cannot access the layer 0 GetPartTable function directly as this breaks the integrity of the three layer structure. This function returns a structure with some of the information from the partition tables for each disk. The information is returned in a nicer format, more suited to displaying to an end user by the calling application. The form of the PARTINFO type is shown below:

```
typedef struct partition_nice {
    unsigned uDisk;
    unsigned uPart;
    unsigned uPartType;
    char sPartType[11];
    unsigned long ulSize;
} PARTITION;

typedef struct partinfo {
    unsigned short uParts;
    PARTITION far *lpPartTbl;
} PARTINFO;
```

As can be seen the partition\_nice structure has elements for the disk drive and partition number. These are in the more user friendly format of 1 is the 1st HDD or Partition, 2 the second, etc. The uPartType is the hex value for the partition type and sPartType is a small string describing the partition. The supported partition types are shown below:

<u>uPartType</u>	<u>sPartType</u>	<u>Partition Type</u>
0x05	DOSEXT	DOS Extended Partition
0x06	DOS	DOS Primary Partition
0x83	LINUXSWP	Linux Swapfile Partition
0x82	LINUX	Linux filesystem Partition

The ulSize element gives the size of the partition in bytes.

The elements of the partinfo structure are the number of PARTITION entries in the returned table, and a pointer to the location in memory containing the partinfo structure.

## 5.20 Layer 2: Notes

The layer 2 functions have access to two global variables that keep track of the current state of the filesystem:

- ulCurDir - This holds the inode number of the current directory.
- sCurPath - This is a string that holds the current path.

All the functions in this layer are exported from the library by specifying the `__export` keyword in their function definitions. They make up the interface between the ext2lib DLL and the calling application.

## 5.21 Layer 2: The `get_part_info` Function

This function is used by the calling application to return information regarding the partitions found on all the hard disks connected to the PC. It simply calls the layer 1 function `GetPartInfoNice` and returns the returned `PARTINFO` structure to the calling application. For more details on the `PARTINFO` structure, please refer to the 'Layer 1: The `GetPartInfoNice` Function' section, above.

## 5.22 Layer 2: The `ext2_mount_fs` Function

This function is used by the calling application to mount an ext2 filesystem. It accepts two arguments, the hard disk number and the partition number of the filesystem to be mounted. The function performs the following tasks:

1. Check for sensible drive and partition numbers
2. Check that the specified partition is a Linux partition (type 0x82)
3. Call the `ReadSuperBlock` function to read the superblock into memory.
4. Call the `ReadGroupDesc` function to read the group descriptors into memory

The function returns a structure giving information on the partition that has just been mounted. This information is calculated from information located in the duperblock, so the `GetSuperBlock` function is called to retrieve this information from the memory buffer. The structure of the returned `MOUNTINFO` type is shown below:

```
typedef struct mountinfo {
    unsigned long ulInodes;
    unsigned long ulBlocks;
    unsigned long ulFreeInodes;
    unsigned long ulFreeBlocks;
    unsigned long ulBlockSize;
    char sMountTime[26];
    unsigned short usState;
} MOUNTINFO;
```

The elements of this structure hold the following values:

- `ulInodes` - The total number of inodes in the filesystem
- `ulBlocks` - The total number of blocks in the filesystem
- `ulFreeInodes` - The number of free inodes in the filesystem.
- `ulFreeBlocks` - The number of free blocks in the filesystem.
- `ulBlockSize` - The block size in bytes.
- `sMountTime` - A string giving the last time the filesystem was mounted read/write
- `usState` - The current state of the filesystem. This can take one of two values:
  - `EXT2_VALID_FS 0x0001` - Valid Filesystem
  - `EXT2_ERROR_FS 0x0002` - The filesystem may contain errors

The function sets the current path inode, `ulCurDir` to `EXT2_ROOT_INO`, currently 2, and the current path, `sCurPath` to `"/"` then returns the `MOUNTINFO` structure.

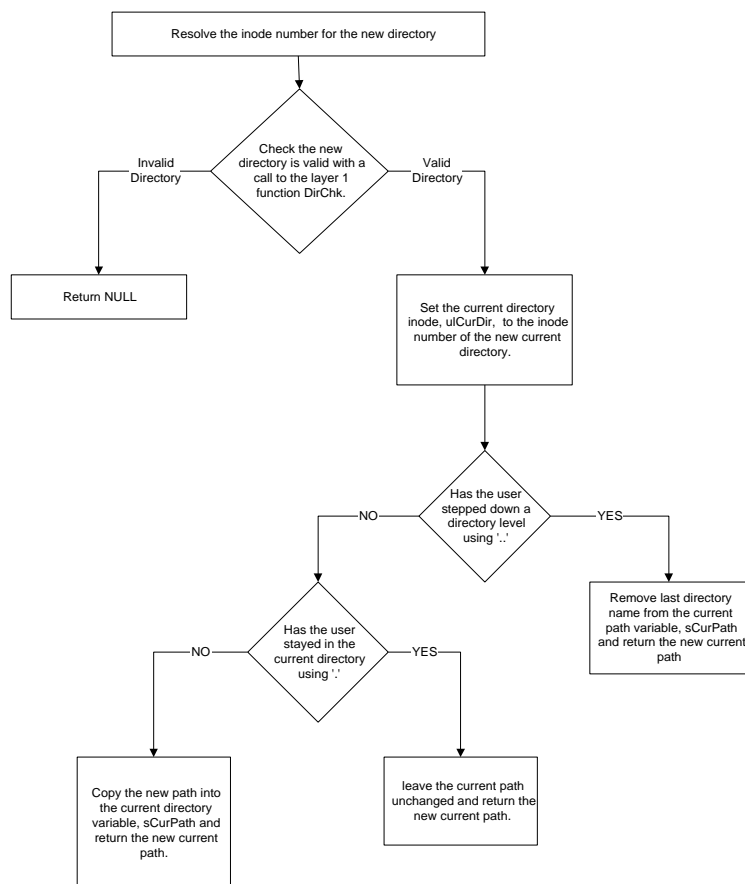
### 5.23 Layer 2: The `ext2_cd` Function

This function is used to change the current directory. It takes one argument, the path of the directory to change to. The supplied pathname has to be added to the end of the current path to give the complete pathname. The function first performs some checking on the supplied path to ensure that there is only one directory separator, `'/'`, separating the current path and the supplied path when they are joined.

The tasks shown in the diagram below then need to be performed by the function. They basically consist of setting the new current directory inode value, `ulCurDir` and setting the current path name variable, `sCuPath`. The first two entries of any `ext2fs` directory are `'.'` and `'..'` which point to the current directory and the previous directory, respectively. If the calling application has called `ext2_cd` with either of these directories as an argument, the current path variable

has to be altered accordingly. The function returns the current path to the calling application.

The structure of the ext2\_cd function:



### 5.24 Layer 2: The ext2\_Is Function

The purpose of this function is to provide the calling application with a directory listing. It takes one argument, the path name of the directory to list. The supplied pathname has to be added to the end of the current path to give the complete pathname. The function first performs some checking on the supplied

path to ensure that there is only one directory separator, '/', separating the current path and the supplied path when they are joined. If no pathname is specified as an argument, the current directory is listed.

The function then simply performs two calls to the layer 1 PathToInode and ReadDir functions to resolve the inode number for the directory and to read the directory, respectively. The DIRECTORY structure returned by the ReadDir function is then returned on to the calling application. For information on the DIRECTORY structure and the structure of the directory listing, please see section 5.16, entitled 'Layer 1: The ReadDir Function'.

### **5.25 Layer 2: The ext2\_get\_inode Function**

This function is used by the calling application to obtain the inode pointed to by the inode number, ullNode, passed to the function. The calling application may wish to use this function to obtain information (type, date, etc.) about a particular file or directory. The function simply calls the layer 1 function GetInode which returns the inode (type INODE). This is then returned to the calling application. For information on the structure of the inode, please see section 3.3.5

### **5.26 Layer 2: The ext2\_cp Function**

The calling application uses this function to copy a file from the ext2 filesystem to a DOS filesystem. The function accepts four arguments:

- sSrcPath - The path of the source file.
- sDestPath - The path of the destination file.
- fOWrite - Overwrite Flag (TRUE = force overwrite, FALSE = do not overwrite)
- Spd - A pointer to a double precision variable to store the average transfer speed for the operation.

The function first performs a similar operation on the source path name as the ext2\_ls and ext2\_cp function, joining the current path, etc. It then makes a call to PathToInode to resolve the inode number of the source file. Once this has been done the layer 1 function CopyFile is called, which performs the file transfer.

The average transfer speed of the file is calculated, so that performance can be monitored. This is achieved by using ftime to record the exact time before the call to CopyFile, and the time when execution returns to the ext2\_cp function. The ftime function fills a small structure, timeb, defined in timeb.h:

```

struct timeb {
    long time ;
    short millitm ;
    short _timezone ;
    short dstflag ;
};

```

The two values that are used are time and millitm which hold the current time, in seconds since 00:00:00 GMT Jan 1<sup>st</sup> 1970, and the fractional part of a second in milliseconds, respectively. The time that it took to copy of the file is calculated by

subtracting the two recorded times. This is then used to calculate the average transfer time by dividing the file size by the time taken to copy it.

The ext2\_cp function returns an error code:

<u>Error Code</u>	<u>Value</u>
No Error	0
INODE_NOT_FOUND	1
BLK_LIST_ERR	2
OPEN_FILE_ERR	3
FILE_EXISTS	4
PATH_NOT_FOUND	5

### 5.27 The Cmdlin User Interface

The cmdlin user interface was developed alongside ext2lib for testing and debugging purposes. It now forms the simple user interface used to demonstrate the functions of ext2lib. It provides a very simple 'command line in a window' that can be used to mount and navigate an ext2 filesystem. It was constructed in Borland C's EasyWin format. This allows a simple windows application to be programmed as if it were a DOS application, however Borland C will compile the application to run under Windows. This route was chosen because cmdlin needed to be developed rapidly to test and debug ext2lib.

The source code for cmdlin is contained in the file cmdlin.c. Examining the source shows a simple structure implementing the ext2lib functions to give command line output. The main event loop is provided by a switch command which executes small functions to perform the help, part, mount, cd, ls and cp commands. The code is self-explanatory.

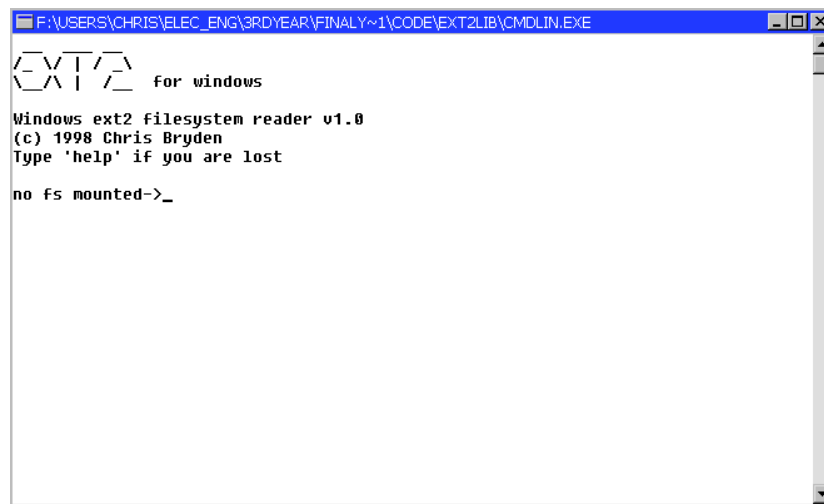
For instructions on using the cmdlin interface, please see appendix D

For more information on implementing the ext2lib.dll library in application programs, please see the 'Ext2lib Programmers Implementation Guide' in appendix E.

## 6. The Cmdlin User Interface

The user interface supplied with ext2lib.dll is a simple command line based application. It is mainly included for demonstration and implementation reference purposes and is not intended to be a complete application in itself, although it does provide a fully functional way of browsing a local ext2 filesystem and copying files from it. For instructions on using cmdlin, please see appendix D

On starting cmdlin, a window is displayed, similar to that shown below:



```

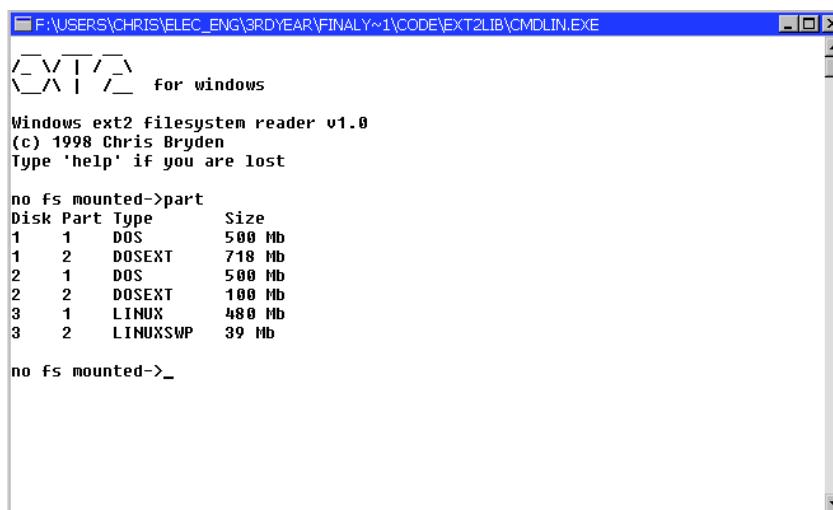
F:\USERS\CHRIS\ELEC_ENG\3RDYEAR\FINALY~1\CODE\EXT2LIB\CMDLIN.EXE
┌─┴─┐
└─┬─┘ for windows

Windows ext2 filesystem reader v1.0
(c) 1998 Chris Bryden
Type 'help' if you are lost

no fs mounted->_
  
```

A command prompt is displayed, and the user can enter a command:

The part command shows the partitions found on the hard disks connected to the PC. It displays a table similar to the following:



```

F:\USERS\CHRIS\ELEC_ENG\3RDYEAR\FINALY~1\CODE\EXT2LIB\CMDLIN.EXE
┌─┴─┐
└─┬─┘ for windows

Windows ext2 filesystem reader v1.0
(c) 1998 Chris Bryden
Type 'help' if you are lost

no fs mounted->part
Disk Part Type      Size
1 1 DOS              500 Mb
1 2 DOSEXT           718 Mb
2 1 DOS              500 Mb
2 2 DOSEXT           100 Mb
3 1 LINUX            480 Mb
3 2 LINUXSMP         39 Mb

no fs mounted->_
  
```

This shows the disk and partition number, the type and the size of the partition. The values for partition and drive can then be used as arguments for the mount command to mount a ext2 partition.

On successfully mounting a partition, the mount command displays some information regarding the mounted filesystem:

```

F:\USERS\CHRIS\ELEC_ENG\3RDYEAR\FINALY~1\CODE\EXT2LIB\CMDLIN.EXE
\X\T/\ for windows
Windows ext2 filesystem reader v1.0
(c) 1998 Chris Bryden
Type 'help' if you are lost

no fs mounted->mount 3 1
Attempting to mount partition....

Success....

Mounted ext2 filesystem on HDD 3, Partition 1
FileSystem Size 503676928 bytes (480 Mb)
Free Inodes 93707/122976
Free Blocks 60185/491872
Last Mounted on Wed Mar 18 12:58:10 1998

Filesystem is clean, Have a nice day :)

3,1:/->

```

The filesystem can now be navigated by using the ls and cd commands and files copied using the cp command. The current drive, partition and directory are shown in the command prompt:

```

F:\USERS\CHRIS\ELEC_ENG\3RDYEAR\FINALY~1\CODE\EXT2LIB\CMDLIN.EXE
tmp
var
bin
dev
etc
lib
sbin
usr
boot
home
mnt
proc
root
.automount

3,1:/->cd /usr/src

3,1:/usr/src/>ls
.
..
linux-2.0.27
linux
redhat

3,1:/usr/src/>

```

## 7. Analysis

This section provides an assessment of the success of the project. Several criteria are used to analyse the project, a comparison with the original specification, reliability and performance.

The original specification specified that the application should read and write the ext2 filesystem, this was very much a long term goal, and as more research was conducted it became apparent that to code filesystem writing was beyond the scope of the project. Learning about the ext2 filesystem was a project aim, the code that makes up the application was written from scratch. However, to write a read/write application the best approach would be to port the relevant kernel source to Windows. This is a very different project, and is also not a practical proposition in the given timeframe. The ext2lib library has been designed with expansion in mind. It would be possible, given enough time, to add extra layer 1 functions to enable the library to write to the partition.

The main thrust of the project has been to produce the ext2lib library. The user interface is another complete program, using the ext2lib library to access the ext2 filesystem. The user interface that is supplied with ext2lib is a simple command line interface, not the Windows Explorer style interface from the specification. There was simply insufficient time for the development of the Explorer user interface, however, because the ext2 reading functions are contained in a library, a user interface such as this can quite easily be added. It is due to the fact that the project has produced a dynamic library that can be used by many programs that this flexibility exists to expand the application. The result of the project has to been to produce a component that can then be incorporated by other programs to produce a complete application. The user interface that has been supplied is a simple command line one, and although it can be used very successfully to read an ext2 filesystem, it is mainly intended to be a demonstration tool and an easy to follow implementation example.

The program has been thoroughly tested on the development machine with various different ext2 filesystem installations. However, the shortage of the necessary hardware has meant that testing on other machines has been somewhat limited. To test the library various files of different types and sizes have been copied from the ext2 filesystem to the DOS filesystem, and the ext2 filesystem was navigated thoroughly using the cd and ls commands. The library has been written with compatibility in mind, however. It used the PC standard BIOS routines to read the disk, and has been designed to cope with variations in ext2 filesystem installations. The library's reliability is good when the filesystem contains no errors, and although measures have been employed to prevent program crashes, the results on a filesystem containing errors is unpredictable. The faults that have been encountered are usually 'General Protection

Exceptions 'or 'Divide by Zero' errors, which do not lock the machine, but merely result in the application being terminated. It should be noted though, that these faults only occur if the filesystem contains errors. On a clean filesystem very few errors occur.

The copy file function returns an average transfer speed when copying a file and this can be used for measuring file transfer performance. The performance of the cd and ls commands is not measured in this way, but a feel for their performance is quickly gained by navigating the filesystem a little. The overall performance of the library is good, the ls command produces directory listings more or less instantly and the cp command copies files with very reasonable average transfer times, up to about 300 Kb/sec. This means that a 5MB file copies from the ext2 partition to the DOS partition in about 15-20 seconds. The performance of these functions is helped considerably by the read-ahead cache that is implemented in the library. The cd command sometimes pauses for no more than about 0.5 sec before returning the command prompt while it is reading the disk. However, the FIFO cache of recently read blocks has improved this.

Overall the project has been successful within it's scope. A library containing the functions necessary to mount and read an ext2 filesystem (cd, ls and cp) along with a Windows based command line interface have been produced that work, are reliable and perform reasonably well.

## 8. Conclusions

The result of the project has been to develop an MS Windows dynamic library containing functions that can be used to navigate and copy files from a hard disk partition that contains the Second Extended Filesystem (ext2fs). Also, an application that runs under MS Windows 95 or 3.x that uses the above library to implement a command line interface has been produced.

The specification for the project was designed in such a way as to offer several fall-back positions to ensure that if all the aspects of the original specification proved impractical to implement in the given time frame, a useful program was still produced. The program that was produced differs in two main ways from the specification, it is read only and it has no full graphical user interface. These were not developed due to time constraints. However, the program that was produced still provides a useful, and until now, largely unavailable, method of reading files located on an ext2fs partition.

The library's three layer structure provided a framework on which to base the workload for the project. Layers 0 and 1 are the basis to the whole project, and therefore it was critical that these were developed first. It was envisaged that each of these layers would require only a month to develop, with a further two months for the implementation of layer 2 and the graphical user interface. However, the technical difficulty and therefore amount of time necessary to implement layers 0 and 1 was underestimated in the original specification and plan of action, and it is for this reason that there was no time to implement filesystem writing and the graphical user interface.

Layer 0 achieves the task of accessing hard disks that are not normally visible to the user under Windows. This has involved using interrupts to access the PC BIOS to perform operations on the hard disk. To implement these functions an understanding of interrupts, implementing the interrupt calls in assembly language and the PC BIOS hard disk functions had to be gained. Also, implementation involved learning how to use the DOS Protected Mode Interface (DPMI) to issue real-mode interrupts in order to call the BIOS disk functions from within a Windows application.

The raw information read from the disk is then translated by layer 1 into information useful to the user, such as directories and files. This is then used to implement familiar filesystem commands such as 'ls', 'cd' and 'cp'. In order to write the code for this section of the library, a thorough understanding of the inner workings of the ext2 filesystem and how it was laid out on the hard disk had to be gained. This involved a long process of research and experimentation culminating in the writing of a set of functions able to manipulate inodes, group

descriptors and the superblock, not to mention files and directories, in order to implement the filesystem commands.

As a read-only method of accessing an ext2 filesystem the library still has deficiencies. Due to its reliance on DPML to access the PC BIOS directly, it will not work under Windows NT. Also, its use of DPML limits it to being coded as a 16 bit library, rather than the more efficient 32 bit programs used by Windows 95. This restriction to a 16 bit library has also meant that the long filenames used by Windows 95 have not been supported. The C code used to implement the library is not ideal in style and some rough edges still remain, especially in the area of memory management. This is due to the fact that the project provided an exercise whose main benefit was learning how to program in C. The code style represents this learning process, with the implementation improving as the project progressed. Unfortunately time did not permit re-coding of dubious sections with the benefit of the knowledge gained during the project. The result of this is that reliability and performance are not optimal.

Allowed further time there are many developments that could be made to the project. Obviously, write enabling the library would be a huge advantage. However this is a technically difficult proposition that would require a long development time and major additions to the library. Also the library does not currently handle symbolic links, correcting this is a smaller task, though. An improvement that could be achieved without changing the library in any way is implementing a graphical user interface. The best solution would be to write an application that embeds the functions performed by the library within Windows Explorer. This would mean that a mounted ext2 filesystem would appear as a read-only disk within Explorer, enabling files to be dragged and dropped into directories on the FAT/VFAT disks that are already visible to Explorer as standard.

To conclude, the project has resulted in a genuinely useful and extendible program, however, this seems a secondary achievement compared to the amount that was learned about the ext2 filesystem (and filesystems in general), programming for MS Windows in C and the requirements necessary to manage a large project.

## 9. References

[1] The DOS Interrupt List,  
Ralph Brown 1997,  
<http://www.pobox.com/~ralf/files.html>

[2] The Extended-2 Filesystem Overview,  
Gadi Oxman 1995,  
<http://ftp.cs.umn.edu/pub/Linux/sunsite/system/filesystems/ext2/Ext2fs-overview-0.1.ps.gz>

[3] Design and Implementation of the Second Extended Filesystem,  
Remy Card,  
Theodore Ts'o,  
Stephen Tweedie ,  
<http://www.redhat.com:8080/HyperNews/get/fs/ext2intro.html>

[4] The Ext2fs Source  
Remy Card  
[/usr/src/linux/fs/ext2/](http://usr/src/linux/fs/ext2/)

## 10. Bibliography

Programmer's Technical Reference for MSDOS and the IBM PC,  
Dave Williams 1997,  
ISBN 1-878830-02-3

The Microsoft website  
<http://www.microsoft.com>

The RedHat website  
<http://www.redhat.com>

Linux Online  
<http://www.linux.org>

Dr Linux  
Linux Systems Labs  
ISBN 1885329-05-9

C By Dissection  
Al Kelley & Ira Pohl 1992  
ISBN 0-8053-3140-9

## 11. Appendix A:

## 12. Final Year Project Proposal

### 12.1 Introduction

It has become increasingly common for personal computers to be run on a dual boot system between Linux and Windows 95. Linux is capable of reading the VFAT file system used by windows 95, however Windows 95 does not have the capability to read a hard disk partition formatted with the ext2 file system used by a modern Linux installation. It would be beneficial, therefore, to have a means of reading and writing an ext2 partition from within Windows 95.

### 12.2 Project Aims

#### 12.2.1 Phase One

To produce a Windows 95 utility for file transfers to and from a local ext2 partition.

#### 12.2.2

#### 12.2.3 Phase Two (If Time Permits)

To produce a utility that makes an ext2 partition appear as a VFAT partition to Windows 95, allowing reading and writing from within Windows applications such as Explorer.

### 12.3

### 12.4 Project Requirements

- An IBM Compatible PC with Windows 95 / Linux dual boot installed. I can provide this.
- A C / C++ Development environment (e.g. Borland C++). I can provide this
- (Possibly) Visual Basic for user interface work. I can provide this.

## 13. Appendix B:

## 14. Final Year Project - Using Linux Filesystems under Windows

### 14.1 Initial Specification (5<sup>th</sup> October 1997)

This document provides an initial specification for a standalone Windows application used to transfer files to and from an ext2 partition. The details are a preliminary specification and are subject to change.

#### 14.1.1 Program Structure

It has been decided that the most versatile way of creating the application is to write a Windows library (.DLL) to handle the reading and writing to the ext2 partition. This approach has been chosen because the DLL can be written in C and the user interface for the application can be written in a language more suitable to rapid UI development, such as visual basic. The DLL also is reusable if a new interface was to be designed for phase two of the project.

#### 14.1.2 The Windows Library (DLL)

The library is to provide the following functions.

- Reading of directory listing on the Linux filesystem, complete with file attributes and time stamp.
- Transfer of a specified file to the calling program.
- Transfer of a specified file from the calling program to the ext2 partition.
- Deletion of files on the ext2 partition

In order to provide the following functions the DLL must contain:

- Code to read the physical disk containing the ext2 partition. The desired function would accept a physical disk sector as a parameter and return the data held on that sector.
- Functions to read the partition table of a disk and determine the areas on the disk occupied by the ext2 filesystem.
- Code to interpret the superblock information for the ext2 filesystem to be able to retrieve and write data from and to the correct places on the disk using the physical disk reading function mentioned above.

### 14.1.3 The User Interface

Visual Basic 5 is the language that has been chosen to write the user interface code. The basic design for the user interface will be modelled on the Windows Explorer. That is the application window will consist of two panes, the left hand pane showing a directory tree and the right hand pane showing a listing of the contents of the currently selected directory. The application is to be drag and drop compliant with the Windows Explorer, enabling files to be dragged between the ext2 explorer to the Windows explorer (this includes areas such as the desktop).

## 15. Appendix C:

## 16. Plan of Action as of 5<sup>th</sup> October 1997

### 16.1 Approach

The first objective is to produce the windows library containing all the functions necessary to read the ext2 partition. Reading the disk takes priority over writing to the disk and this will be the first objective. Constructing functions to perform file writes to the ext2 partition is a secondary objective and its feasibility in the given time span will be gauged on further research.

### 16.2 October 1997

- Research BIOS routines for reading disks not normally visible to DOS. Borland C++ 4.5 has functions provided in bios.h for achieving this from a DOS application, but they are not portable to Windows applications. The desired aim is the construction of a function to read or write data to a specified physical sector on a fixed disk.
- Research the structure of partition tables for fixed disks with the aim of constructing a function to return the location of an ext2 partition.
- Research the structure of the ext2 filesystem. The aim of this is to produce a set of functions to read the directory structure of the filesystem, return the physical locations on the disk that a specified file occupies, return the location of free blocks on the disk, determine disk free space values and perform the housekeeping tasks involved in writing to the disk.

### 16.3 November and December 1997

- Produce working library functions to perform the operations researched above. The objective is to produce at least a library capable of performing the functions necessary to read the ext2 filesystem by the beginning of the second semester. Reassess the viability of creating an application capable of writing to the Linux filesystem.
- Running alongside the development of the library, test programs will be developed in VB to aid in the development of the library.

### 16.4 January and February 1998

- Wind up work on the library file and design and construct the user interface for the application. Project completion is aimed for Easter 1998.

## 17. Appendix D:

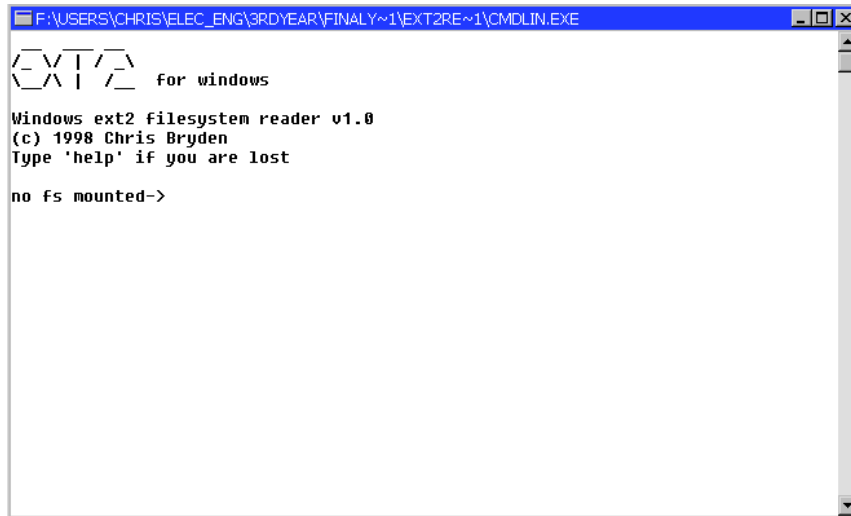
## 18. Cmdlin User Manual

### 18.1 Installation

To install Ext2lib, unzip the files cmdlin.exe, ext2lib.dll and bc450rtl.dll from ext2lib\_1.01.zip to a new directory. Cmdlin is now ready for use

### 18.2 Starting Cmdlin

To start cmdlin, simply double click on it's icon in Windows Explorer. This will open the cmdlin window, and present the 'no fs mounted' command prompt.

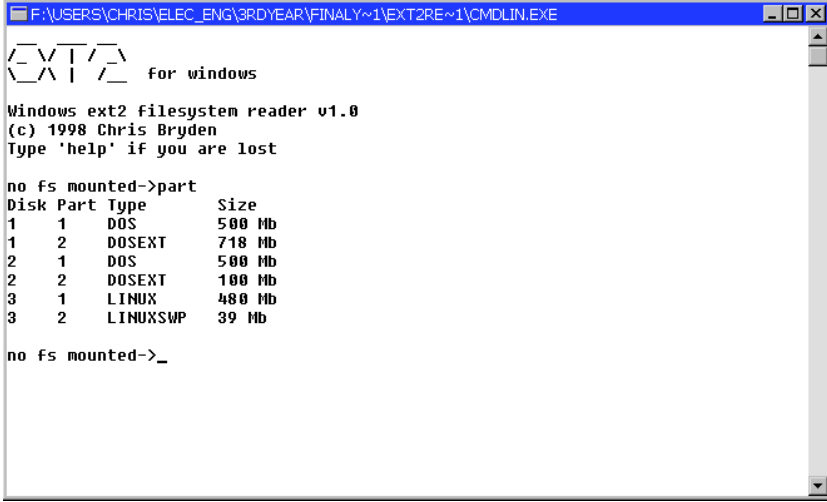
A screenshot of a Windows command prompt window titled "F:\USERS\CHRIS\ELEC\_ENG\3RDYEAR\FINALY~1\EXT2RE~1\CMDLIN.EXE". The window contains the following text:

```
EXT2  for windows
Windows ext2 filesystem reader v1.0
(c) 1998 Chris Bryden
Type 'help' if you are lost
no fs mounted->
```

The cmdlin application is now ready to mount an ext2 filesystem.

### 18.3 Viewing Hard Disk Partition Information - The 'part' command

To give the hard disk and partition numbers required by the 'mount' command, the 'part' command is used. It displays all the hard disk partitions found on the system. Simply type part and hit return, and a display similar to the following is given:



```
F:\USERS\CHRIS\ELEC_ENG\3RDYEAR\FINALY~1\EXT2RE~1\CMDLIN.EXE
[EXT2] for windows
Windows ext2 filesystem reader v1.0
(c) 1998 Chris Bryden
Type 'help' if you are lost

no fs mounted->part
Disk Part Type      Size
1      1      DOS        500 Mb
1      2      DOSEXT     718 Mb
2      1      DOS        500 Mb
2      2      DOSEXT     100 Mb
3      1      LINUX      480 Mb
3      2      LINUXSWP   39 Mb

no fs mounted->_
```

### 18.4 Mounting an Ext2 Filesystem - The 'mount' command

In order to read files from an ext2 partition, it must first be mounted. TO mount a partition, the 'mount' command is used. It takes two arguments from the command line: The hard disk number and partition number of the partition containing the ext2 filesystem.

e.g. mount 3 1 will mount the filesystem on hard disk 3, partition 1.

If the partition is mounted successfully, some information is given about the filesystem, and the command prompt is returned.

```

F:\USERS\CHRIS\ELEC_ENG\3RDYEAR\FINALY~1\EXT2RE~1\CMDLIN.EXE
Type 'help' if you are lost

no fs mounted->part
Disk Part Type      Size
1 1  DOS             500 Mb
1 2  DOSEXT          718 Mb
2 1  DOS             500 Mb
2 2  DOSEXT          100 Mb
3 1  LINUX           480 Mb
3 2  LINUXSWP        39 Mb

no fs mounted->mount 3 1
Attempting to mount partition....

Success....

Mounted ext2 filesystem on HDD 3, Partition 1
FileSystem Size 503676928 bytes (480 Mb)
Free Inodes 93707/122976
Free Blocks 60185/491872
Last Mounted on Wed Mar 18 12:58:10 1998

Filesystem is clean, Have a nice day :)

3,1:/->_

```

If the specified partition does not contain a valid ext2 filesystem, an error is given. The cmdlin will still mount a partition even if it has been marked as dirty, although this is not recommended as program behaviour may be unpredictable.

Once the partition has been mounted, the following command are available to the user.

### 18.5 Viewing Directory Contents - The 'ls' Command

To view the contents of the current directory, simply type ls and hit return. The dirtory listing is displayed as shown below:

```

F:\USERS\CHRIS\ELEC_ENG\3RDYEAR\FINALY~1\EXT2RE~1\CMDLIN.EXE
Filesystem is clean, Have a nice day :)

3,1:/->ls
.
..
lost+found
apps
data
win95
tmp
var
bin
dev
etc
lib
sbin
usr
boot
home
mnt
proc
root
.automount

3,1:/->

```

To view the contents of a directory other than the current directory, a path can be supplied to ls. For example,

ls /usr/bin/ Gives a directory listing of the /usr/bin directory.

### 18.6 Changing Directory - The 'cd' Command

The cd command is used to change the current directory. The current directory is displayed in the command prompt. The cd command accepts one argument, the directory to change to. For example,

cd usr changes the current directory to the 'usr' directory.

```
3,1: /->cd usr
3,1: /usr/>cd src
3,1: /usr/src/>
```

### 18.7

### 18.8 Copying a File - The cp Command

The cp command is used to copy a file from a location on the ext2 filesystem to a location on the DOS filesystem. It accepts two arguments, the path of the source file and the path of the destination file. So, for example,

cp /usr/src/linux/Makefile F:\Users\Chris\code\Makefile copies the file 'Makefile;' from the ext2 filesystem to the DOS directory shown.

Note that both paths must be specified in full, with the drive letter and the destination filename specified for the destination path.


If the destination file already exists, a prompt appears asking whether to overwrite the file:

```
3,1: /usr/src/linux-2.0.27/>cp Makefile f:\users\chris\Makefile
The file 'f:\users\chris\Makefile' exists, overwrite?
```

Simply press 'y' to overwrite the file, or 'n' to abort the operation.

If a file is copied successfully, a message is displayed indicating this and giving the average transfer rate. If an error occurs an error message is displayed.

## **18.9 Closing Cmdlin**

To quit cmdlin, type 'exit' at the command prompt, and close the Window using the  icon at the top left hand corner of the Window.

## 19. Appendix E:

## 20. Programmers Implementation Guide

### 20.1 Introduction

This guide is intended to provide an outline for programmers wishing to use the ext2lib library to incorporate ext2 file reading capabilities into an application. It provides an outline of how to implement each of the exported functions in a target application, and example code in C is given.

Included with the ext2lib library are two header files that contain definitions, data structures and types and function prototypes for the library. These are:

- Ext2lib.h - Header information for the ext2lib library
- Ext2\_fs.h - Linux source file containing header information for the ext2 filesystem. © Remy Card

Also an import library ext2lib.lib is included. Some of the types defined in windows.h are used in the example code.

The library provides two commands that are used in the process of mounting a filesystem. The first is get\_part\_info, which is used to return partition information for the disks connected to the PC. This is used to inform the user of the mountable partitions. The next function used in the process is the actual ext2\_mount\_fs function. This is used to actually mount the partition.

Once the partition has been mounted, four functions control the manipulation of the filesystem. These are the commands ext2\_ls, ext2\_cd and ext2\_cp which perform the functions that would be expected of the ls, cd and cp commands at a command line. A function called ext2\_get\_inode is also included. This is used to retrieve the inode for a specified file.

### 20.2 Implementing the get\_part\_info Function

```
PARTINFO FAR PASCAL get_part_info(void);
```

This function accepts no arguments, and returns a small structure with the type, PARTINFO. This is defined in ext2lib.h:

```
typedef struct partinfo {
    unsigned short uParts;
    PARTITION far *lpPartTbl;
} PARTINFO;
```

- The elements of this structure provide the following
- uParts is the total number of partitions found
  - lpPartTbl is a far pointer to the location in memory of the partition table

The partition table held in memory is made up of a sequence of the following PARTITION entries, each representing a partition on disk.

```
typedef struct partition_nice {
    unsigned uDisk;
    unsigned uPart;
    unsigned uPartType;
    char sPartType[11];
    unsigned long ulSize;
} PARTITION;
```

As can be seen the partition\_nice structure has elements for the disk drive and partition number, uDisk and uPart. These are in the format of 1 is the 1st HDD or Partition, 2 the second, etc. The uPartType is the hex value for the partition type and sPartType is a small string describing the partition. The supported partition types are shown below:

<u>uPartType</u>	<u>sPartType</u>	<u>Partition Type</u>
0x05	DOSEXT	DOS Extended Partition
0x06	DOS	DOS Primary Partition
0x83	LINUXSWP	Linux Swapfile Partition
0x82	LINUX	Linux filesystem Partition

The ulSize element gives the size of the partition in bytes.

These structures are used to display the partition information to the user, the implementation is straightforward as the following example shows:

```
void DisplayPartitonInformation(void)
{
    PARTINFO PartInfo;
    PARTITION Partition;
    unsigned i;

    PartInfo = get_part_info();

    for(i=0;i<=(PartInfo.uParts-1);++i)
    {
        _fmemcpy(&Partition, PartInfo.lpPartTbl + i, sizeof(PARTITION));
        if(Partition.ulSize != 0)
        {
            /* CODE TO DISPLAY ENTRIES GOES HERE */
        }
    }

    return;
}
```

### 20.3 Implementing the `ext2_mount_fs` Function

```
MOUNTINFO FAR PASCAL ext2_mount_fs(unsigned short bDrive,
                                     unsigned short bPart);
```

This function accepts two arguments, the hard disk number and partition number of the partition containing the filesystem to mount. These are in the same format as displayed by the `get_part_info` function, above.

This function returns a small structure of the type `MOUNTINFO`. This contains some information that the calling application may wish to display to the user:

```
typedef struct mountinfo {
    unsigned long ulInodes;
    unsigned long ulBlocks;
    unsigned long ulFreeInodes;
    unsigned long ulFreeBlocks;
    unsigned long ulBlockSize;
    char sMountTime[26];
    unsigned short usState;
} MOUNTINFO;
```

The elements of this structure hold the following values:

- `ulInodes` - The total number of inodes in the filesystem
- `ulBlocks` - The total number of blocks in the filesystem
- `ulFreeInodes` - The number of free inodes in the filesystem.
- `ulFreeBlocks` - The number of free blocks in the filesystem.
- `ulBlockSize` - The block size in bytes.

- sMountTime - A string giving the last time the filesystem was mounted read/write
- usState - The current state of the filesystem. This can take one of two values:
  - EXT2\_VALID\_FS 0x0001 - Valid Filesystem
  - EXT2\_ERROR\_FS 0x0002 - The filesystem may contain errors

If the MountInfo structure contains valid data on the return of the ext2\_mount\_fs function call, the filesystem was mounted successfully.

This ext2\_mount\_fs function is simple to implement, as shown in the example below.

```
void MountExt2Filesystem(unsigned short Drive, unsigned short Part)
{
    MOUNTINFO MountInfo;

    _fmemset(&MountInfo, 0, sizeof(MOUNTINFO));

    MountInfo = ext2_mount_fs(Drive, Part);

    if(MountInfo.ulInodes == 0)
    {
        /* NOT A VALID EXT2 FILESYSTEM */
    }
    else
    {
        /*PARTITION MOUNTED SUCCESSFULLY, DISPLAY MOUNTINFO TO USER */
        /*CHECK usState TO DETERMINE CLEAN OR DIRTY FILESYSTEM */
    }
    return;
}
```

## 20.4 Implementing the ext2\_ls Function

```
DIRECTORY FAR PASCAL ext2_ls(char far PathName[MAX_PATH_LEN]);
```

This function accepts one argument, the address of a string containing the path of the directory to list, if this path is not specified, the current directory is listed. The function returns a small structure of type DIRECTORY containing two elements:

```
typedef struct directory {
    unsigned long ulDirLen;
    LPBYTE lpDirData;
} DIRECTORY;
```

ulDirLen is the directory list length in bytes and lpDirData is a far pointer to the memory location that the directory list is stored in.

The directory list held in the memory location pointed to by lpDirData is stored in the same format as directories are stored on the disk. That is, a linked list of directory entries having the following format, defined in ext2\_fs.h :

```
#define EXT2_NAME_LEN 255

struct ext2_dir_entry {
    unsigned long  inode;           /* Inode number */
    unsigned short rec_len;        /* Directory entry length */
    unsigned short name_len;       /* Name length */
    char           name[EXT2_NAME_LEN]; /* File name */
};
```

The directory entries are not of the same size, as the next entry starts as soon as the name of the previous entry ends. For this reason, rec\_len is used to determine the start of the next directory entry. This is shown in the example code below:

```
void ListDirectory(char PathName[MAX_PATH_LEN])
{
    unsigned long ulDirOffset = 0;
    DIRENT DirEnt;
    DIRECTORY Directory;

    Directory = ext2_ls(&PathName);

    if(Directory.ulDirLen == 0 || Directory.lpDirData == NULL)
    {
        /* INVALID DIRECTORY */
    }

    while(ulDirOffset < Directory.ulDirLen)
    {
        /*get directory entry*/
        _fmemcpy(&DirEnt, Directory.lpDirData + ulBuffOffset,
                sizeof(DIRENT));

        /*Increment next entry offset */
        ulBuffOffset += DirEnt.rec_len;

        /*null terminate string*/
        DirEnt.name[DirEnt.name_len] = NULL;

        /*DISPLAY DIRECTORY ENTRY*/
    }
    return;
}
```

## 20.5 Implementing the `ext2_cd` Function

```
char far* FAR PASCAL ext2_cd(char PathName[MAX_PATH_LEN]);
```

This function accepts one argument, the address of a string containing the path to the directory to change the current directory to. On successfully changing the current directory, the function returns the path of the new current directory. This function is extremely simple to implement, as shown in the example code below:

```
void ChangeDirectory(char PathName[MAX_PATH_LEN])
{
    if(ext2_cd(&PathName) == NULL)
        /*INVALID DIRECTORY*/
    else
        /*SUCCESS*/

    return;
}
```

## 20.6 Implementing the `ext2_cp` Function

```
BYTE FAR PASCAL ext2_cp(char sSrcPath[MAX_PATH_LEN],
                        char sDestPath[MAX_PATH_LEN],
                        BOOL fOWrite, double *Spd);
```

This function accepts four arguments, the meaning of each is described below:

- `sSrcPath` - The path of the source file.
- `sDestPath` - The path of the destination file.
- `fOWrite` - Overwrite Flag (TRUE = force overwrite, FALSE = do not overwrite)
- `Spd` - A pointer to a double precision variable to store the average transfer speed for the operation.

The destination path has to be specified in full, in the format:

```
<drive>:/<dir>/<subdir>/<destination filename>
```

This means that any command line completion has to be performed by the calling application. The function returns an error code, defined in `ext2lib.h`:

<u>Error Code</u>	<u>Value</u>
No Error	0
INODE_NOT_FOUND	1

BLK_LIST_ERR	2
OPEN_FILE_ERR	3
FILE_EXISTS	4
PATH_NOT_FOUND	5

On successful return, the average transfer speed of the file is stored in the location pointed to by Spd.

An example of the implementation of the function is shown below:

```
void CopyFile( char sSrcPath[MAX_PATH_LEN],
               char sDestPath[MAX_PATH_LEN])
{
    unsigned short err;
    double Spd;

    err = ext2_cp(sSrcPath, sDestPath, FALSE, &Spd);

    switch(err)
    {
        case FILE_EXISTS:

            /*DESTINATION FILE ALREADY EXISTS*
            PROMPT USER FOR OVERWRITE?
            IF 'Y' CALL EXT2_CP WITH THE FOWRITE FLAG
            SET TO TRUE*/

            break;

        case PATH_NOT_FOUND:
            /*PATH NOT FOUND ERROR HANDLING*/
            break;

        case 0:
            /*FILE COPIED SUCCESSFULLY*/
            break;

        default:
            /*AN ERROR OCCURRED*/
            break;
    }

    return;
}
```

## 20.7 Implementing the `ext2_get_inode` Function

```
INODE FAR PASCAL __export ext2_get_inode(unsigned long ulInode);
```

This function is used to return the inode for the inode number specified as the argument to this function. The inode number can be retrieved from the directory listing. The purpose of this function is mainly to display extra information on files to the user.

It returns an inode structure as defined in `ext2lib.h` and given the type `INODE`:

```
/*  
 * Structure of an inode on the disk  
 */  
struct ext2_inode {  
    unsigned short i_mode;  
    unsigned short i_uid;  
    unsigned long i_size;  
    unsigned long i_atime;  
    unsigned long i_ctime;  
    unsigned long i_mtime;  
    unsigned long i_dtime;  
    unsigned short i_gid;  
    unsigned short i_links_count;  
    unsigned long i_blocks;  
    unsigned long i_flags;  
    unsigned long l_i_reserved1;  
    unsigned long i_block[EXT2_N_BLOCKS];  
    unsigned long i_version;  
    unsigned long i_file_acl;  
    unsigned long i_dir_acl;  
    unsigned long i_faddr;  
    unsigned char l_i_frag;  
    unsigned char l_i_fsize;  
    unsigned short i_pad1;  
    unsigned long l_i_reserved2[2];  
};
```

- `i_mode` - The type of file (character, block, link, etc.) and access rights to the file. This field is best described by representing it as an octal number. Since it is a 16 bit number, there will be 6 octal digits. The rightmost four digits are bitwise fields:

The last three digits (Octal digits 0,1 and 2) are the file permissions, in the form `rw xrwxrwx`. Digit 2 refers to the user, digit 1 to the group and digit 0 to everyone else.

The leftmost two octal digits are used to indicate the type of file that the inode points to:

Note that the leftmost octal digit can only take the value of 0 or 1, since the total number of bits is 16. The type of file is one of:

- 01 - FIFO file.
- 02 - Character device.
- 03 - Directory
- 06 - Block Device
- 10 - Regular File
- 12 - Symbolic Link
- 14 - Socket

- `i_uid` - The user id of the owner of the file.
- `i_size` - The file size in bytes.
- `i_atime` - The time the file was last accessed.
- `i_ctime` - The time the inode information for the file was last changed.
- `i_mtime` - The time the files content was last modified.
- `i_dtime` - The time the file was deleted.
- `i_gid` - The group id of the file
- `i_links_count` - The number of links pointing to the file.
- `i_flags` - This takes one or more of the following values:

`EXT2_SECRM_FL 0x0001` - Secure deletion. If this flag is set, when the file is deleted random data is written in its place.

`EXT2_UNRM_FL 0x0002` - Undelete. If this flag is set and the file is deleted, the system must store enough information for the file to be recovered (provided the space taken by the file has not been overwritten).

`EXT2_COMPR_FL 0x0004` Compress file. The content of the file is compressed. The filesystem code must use compression/decompression algorithms when accessing the file.

`EXT2_SYNC_FL 0x0005` - Synchronous Updates - The inode and indirect blocks are to be written to synchronously only.

Not all of the above features are implemented in the current version of ext2fs.

- `i_reserved1` - Not used.
- `i_block[ EXT2_N_BLOCK ]` - Pointers to blocks allocated to the file represented by this inode.

The inode contains `EXT2_N_BLOCKS`, this is currently 15. Of these block addresses, the first `EXT2_NDIR_BLOCKS` (currently 12) are direct pointers to data blocks. The following entry points to a block of pointers to data blocks (indirect blocks). The entry after points to a block of pointers to blocks of pointers to data blocks (double indirect blocks) and the final entry points to a block of pointers to blocks of pointers to blocks of pointers to data blocks (triple indirect blocks).

- `i_version` - The version of the file, used by NFS.
- `i_file_acl` - The Access control list of the file (not used).
- `i_dir_acl` - The Access control list for the directory (not used).
- `i_faddr` - The block where the fragment of the file resides.
- `i_frag` - The number of fragments in the block.
- `i_size` - The size of the fragment.
- `i_pad1` - padding
- `i_reserved2` - Not used.

An example of the implementation of this function is given below:

```
void DisplayInode(unsigned long InodeNum)
{
    INODE Inode;

    Inode = ext2_get_inode(InodeNum);

    /*DISPLAY INODE INFORMATION TO USER */

    return;
}
```