Long Filename Specification

by vinDaci
fourth release
First Release: November 18th, 1996
Last Update: January 6th, 1998 (Document readability update)

Compatibility

Long filename (here on forth referred to as "LFN") design for Windows95 is designed to be 99% compatible with the old DOS 8.3 format. The 1% discripency comes in when old DOS programs can detect the *presence* of LFN (but unfortunately not the LFN itself), which in no way interferes with regular program operations except for perhaps low-level disk utility programs (such as disk error fixing programs, disk optimization programs, anti-virus program, etc.)

DOS 8.3 Filename Background

I trust that anyone who wish to know the details of LFN has at least a small knowledge in DOS 8.3 filename specification. In this document, however, I'll assume you know very little about the 8.3 filename specs, however. What you need to know in order to understand this documentation is that 8.3 filenames are stored in a place on the disk called the *directory table*. This place contains the list of filenames and other information associated with each file, such as the file date, time, size, attributes, etc. (Note: Contrary to some belief, the directory table is *not* the same as the FAT -- e-mail me if you wish to know what FAT is.)

The file attributes, mentioned above, play some big roles in LFN. It is important to note that a file's attributes are may consist of one or more of the following:

`Archive`
`Read-Only`
`System`
`Hidden`
`Directory`
`Volume`

Most programmers are aware of the Archive, Read-Only, System, and Hidden attrbiutes, but for those of you who don't know, please allow me to explain what each of these attributes is/does:

- The *Archive attribute* tells that a file has been backed up (though most programs just ignore this).
- The *Read-Only attribute* keeps a file from accidentally getting overwritten; note that any program can unset this attribute should it know how to detect this attribute and unset it -- that's why it is used just to keep a file from *accidentally* getting overwritten.
- The *System attribute* tells that the file is used for some important operation so it should not be messed with except by the program that created it; any file with this attribute cannot be seen with the `DIR` command except with the `/a` or `/as` arguments in DOS 5 and above.
- The *Hidden* attribute tells that a file should normally be hidden from the user's view. Any file with this attribute cannot be seen with `DIR` command either, except with the `/a` or `/ah` arguments in DOS 5 and above.

And the explanation of the other attributes (the *really* important ones):

- The *Directory attribute* is used to tell that a file is not actually a file but a directory. This type of file contains a pointer to a part of the disk that contains another directory table; this directory table that's pointed to is the subdirectory of the directory that has the pointer. In another words, when you "CD" to that file, you go into the directory table the file points to, making it look as though you are "inside" that directory. In reality, you only switch the directory tables.
- The *volume attribute* too is used to tell that a file is not actually a file. This attribute is used to indicate the volume label of the drive (the name of the disk). A file with this attribute can never be displayed with the `DIR` command. Furthermore, there can be only one file with this attribute on the entire disk, and this file must be in the root directory of the disk.

  The volume attribute has a funny story attached to it -- There not only exists a file with the volume attribute, but a copy of the volume label is also located in the boot sector (the very beginning of the disk that has weird code and disk info on it) as a readable text. But when you view a directory with the `DIR` command, the one that actually gets displayed is the volume attributed file's name, not the volume label in the boot sector. Furthermore, even though files with volume attribute is hidden from the `DIR` command, programs, when retrieving filenames, can retrieve volume labels. All these factors about volume attributes come into a big factor when we look at Long Filenames.

As an addendum, it might be interesting to note that each 8.3 file entry is 32 bytes long but that not all 32 bytes are used to store data -- some of them are plainly left as blank bytes. In Windows95 version of the directory table, however, all 32 bytes are used.

General Specification

Just like DOS 8.3 filenames, Windows95 LFNs are also stored on directory tables, side-by-side with DOS 8.3 filenames. On top of that, to achieve compatibility with old DOS programs Microsoft designed LFN in a way so it resembles the old DOS 8.3 format. Furthermore, an 8.3 format version of LFN (`tttttt~n.xxx`) is also present next to each LFN entry for compatibility with non-LFN supporting programs.

Organization

From a low-level point-of-view, a normal directory table that only contains 8.3 filenames look like this:

...
8.3 entry
8.3 entry
8.3 entry
8.3 entry

If you look at a directory table that contains LFN entries, however, this is what you will see:

...
LFN entry 3
LFN entry 2
LFN entry 1
8.3 entry (tttttt~n.xxx)

Notice that Long Filenames can be pretty long, so LFN entries in a 8.3 directory structure can take

up several 8.3 directory entry spaces. This is why the above file entry has several LFN entries for a single 8.3 file entry.

Despite taking up 8.3 filename spaces, Long Filenames do not show up with the `DIR` command or with any other program, even the ones that do not recognize the LFN. How, then, do LFN entries get hidden from DOS? The answer is quite simple: By giving LFN entries "Read-only, System, Hidden, and Volume" attributes. (If you do not know details about file attributes, read the above text about DOS 8.3 Filename Background.)

A special attention should be given to the fact that this combination of attributes -- Read-only, System, Hidden, Volume -- is not possible to make under normal circumstances using common utilities found in the market place (special disk-editing programs, such as Norton Disk Editor, is an exception.)

This technique of setting Read-only, System, Hidden, Volume attributes works because most programs ignore files with volume attributes altogether, and even if they don't, they won't display any program that has system or hidden attributes set. And since the Read-only attribute is set, programs won't write anything over it. However, you can view "parts" of the LFN entries if any program is designed to show Volume attributed files.

Storage Organization

To understand the LFN storage organization within a directory table, it is important to understand the 8.3 storage organization. This is mainly due to the fact that LFN entries are stored similar to 8.3 filenames to avoid conflicts with DOS applications.

The format of the traditional DOS 8.3 is as follows:

Offset	Length	Value
0	8 bytes	Name
8	3 bytes	Extension
11	byte	Attribute (`00ARSHDV`) `0`: unused bit `A`: archive bit, `R`: read-only bit `S`: system bit `D`: directory bit `V`: volume bit
22	word	Time
24	word	Date
26	word	Cluster (desc. below)
28	dword	File Size

Note: *WORD* = 2 bytes, *DWORD* = 4 bytes

Everything above you should know what they are except perhaps for the cluster. The cluster is a value representing another part of the disk, normally used to tell where the beginning of a file is. In case of a directory, it is the cluster that tells where the subdirectory's directory table starts.

You may not know this, but LFN specification not only added the capability of having longer filenames, but it also improved the capability of 8.3 filenames as well. This new 8.3 filename improvements are accomplished by using the unused directory table spaces (Remember how I told you that 8.3 filenames take up 32 bytes but not all 32 bytes are used? Now it's all used up!) This new format is as follows -- try comparing it with the traditional format shown above!:

Offset	Length	Value
0	8 bytes	Name
8	3 bytes	Extension
11	byte	Attribute (00ARSHDV)
12	byte	NT (Reserved for WindowsNT; always 0)
13	byte	Created time; millisecond portion
14	word	Created time; hour and minute
16	word	Created date
18	word	Last accessed date
20	word	Extended Attribute (reserved for OS/2; always 0)
22	word	Time
24	word	Date
26	word	Cluster
28	dword	File Size

In any case, this new 8.3 filename format is the format used with the LFN. As for the LFN format itself (seen previously) is stored "backwards", with the first entry toward the bottom and the last entry at the top, right above the new 8.3 filename entry.

Each LFN entry is stored as follows:

Offset	Length	Value
0	byte	Ordinal field (desc. below)
1	word	Unicode character 1 (desc. below)
3	word	Unicode character 2
5	word	Unicode character 3
7	word	Unicode character 4
9	word	Unicode character 5
11	byte	Attribute
12	byte	Type (reserved; always 0)
13	byte	Checksum (desc. below)
14	word	Unicode character 6
16	word	Unicode character 7
18	word	Unicode character 8
20	word	Unicode character 9
22	word	Unicode character 10
24	word	Unicode character 11
26	word	Cluster (unused; always 0)
28	word	Unicode character 12
30	word	Unicode character 13

Throughout this entry, you see "unicode characters". Unicode characters are 2-byte long characters (as opposed to ASCII characters that are 1-byte long each) that support not only traditional latin alphabet characters and arabic numbers but they also support the languages of the rest of the world,

including the CJK trio (Chinese, Japanese, Korean), Arabic, Hebrew, etc. Plus you have some space left over for more math and science symbols. Unicode characters are still going through revisions (on their second revision as I am writing, I believe) but Windows95 has left spaces to more fully support unicodes in the future. You can keep up with the Unicode development by visiting the Unicode webpage at www.unicode.org. Note that, in the 2-byte unicode character, the first byte is always the character and the second byte is always the blank, as opposed to having the first byte blank and the second byte blank. There is a perfectly logical explanation for this but it's kind of long to get into at the moment so e-mail me if you are curious. (If you have a computer dictionary, look up "little endian" and it'll explain everything.) For our purposes, though, just treat every other word as an ASCII character as long as the following byte is a blank character. Anyways, notice that, in order to maintain the compatibility with older programs, the attribute byte and the cluster word had to be kept. Because of this, each unicode character had to be scattered throughout the entry.

By now you probably noticed that there is no file information (size, date, etc.) stored in the LFN entry. Any information about the file itself is stored in the 8.3 filename, located below all the LFN entries (as mentioned before).

The checksum is created from the shortname data. The steps/equation used to calculate this checksum is as follows:

Step #	Task
1	Take the ASCII value of the first character. This is your first sum.
2	Rotate all the bits of the sum rightward by one bit.
3	Add the ASCII value of the next character with the value from above. This is your next sum.
4	Repeat steps 2 through 3 until you are all through with the 11 characters in the 8.3 filename.

In C/C++, the above steps look like this:

```
for (sum = i = 0; i < 11; i++) {
    sum = (((sum & 1) << 7) | ((sum & 0xfe) >> 1)) + name[i];
}
```

This resulting checksum value is stored in each of the LFN entry to ensure that the short filename it points to indeed is the currently 8.3 entry it should be pointing to.

Also, note that any file with a name that does not fill up the 8.3 spaces completely leaves a trace of space characters (ASCII value 32) in the blank spaces. These blank spaces do go into the calculation of the checksum and does not get left out of the calculation.

I'm sure you're dying to know what the ordinal field is. This byte of information tells the order of the LFN entries (1st LFN entry, 2nd LFN entry, etc.) If it's out of order, something is wrong! How Windows95 would deal with LFN when such a thing happen is a mystery to me.
An example of how ordinal field work: The first LFN entry, located at the very bottom as we have seen before, has an ordinal field value 1; the second entry (if any -- remember that a LFN doesn't always have to be tens of characters long), located just before the first entry, has an ordinal field value of 2; etc. As an added precaution, the last entry has a marking on the ordinal field that tells that it is the last entry. This marking is done by setting the 6th bit of the ordinal field.

That is basically all there is to long filenames. But before we end this conversation (while we're on the subject of LFN,) I think it would be interesting to note that, since any long filename can be up to 255 bytes long and each entry can hold up to 13 characters, there can only be up to 20 entries of LFN per file. That means it only needs 5 bits (0th bit to 4th bit) of the ordinal field. And with the 6th

bit used to mark the last entry, two bits are left for open usage -- the 5th and the 7th bit. Whether or not Microsoft is going to do anything with these bits -- or why Microsoft used the 6th bit to indicate the last entry instead of 7th or 5th bit and limited the file length to 255 bits -- remains to be a mystery only Microsoft will keep to itself.

Credit

Much the information in this documentation has been gathered from Norton Utilities for Windows95 user's manual. Detailed researches were done using Norton Utilities Disk Edit. The checksum calculation was graciously donated by Jacob Verhoeks to `comp.os.msdos.programmer` Newsgroup as a reply to my request. Apparently the file Mr. Verhoeks used to get me the checksum code, `vfat.txt`, that comes with newer Linux operating systems have some good information on Windows95 LFN. BTW, I just (like, right now) found out that checksum algorithm is also in Ralf Brown's Interrupt List.
