The Simple File System

18 September 2017

Original Design by Brendan Trotter
This documentation and minor additions by Benjamin David Lunt

Version 1.10.rc02

FYSOS and the Simple File System

This document pertains to and is written for the purpose of adding this file system to FYSOS found at:

www.fysnet.net/fysos.html

It adds small changes to some parts of the original file system design and does so for this purpose. Credit for the original design goes to Brendan Trotter and this original design's specification can be found at:

https://www.d-rift.nl/combuster/vdisk/sfs.html

The additions and modifications will be discussed in detail throughout this document. See then end of this document for FYSOS specific items.

A Brief Summary

There is currently no widely accepted format for freely exchanging data on physical media between computers and other devices. Instead people have been using a variety of file systems that are either not widely accepted or subject to technical, licensing, copyright or patent restrictions.

The primary purpose of the Simple File System is to facilitate the free exchange of data on physical media without unnecessary restrictions or complexity. For this reason the Simple File System has been designed in such a way that it is easy to understand and implement.

The Simple File System (SFS) contains a Super Block, a Reserved Area, a Data Block Area, a Free Space Area, and a Metadata area called the Index Data Area. Files are store in sequential blocks leaving no fragmentation, allowing 64-bit addresses and sizes, and allows more than 16,320 bytes for path and file names. However, the SFS does not have features like permissions, symbolic information, or attributes like other files systems. The SFS is intended to be mostly a write-once file system, though a properly written driver could easily and successfully use it as a read and write file system.

Definitions

The following is a list of definitions to help explain and clear up different aspects of this document.

- Block: A block of physical sectors on the media, starting from one sector, incrementing by powers of two.
- LBN: Linear Block Number, a block number starting at the beginning of the volume. This is not to be confused with LBA, Linear Block Address. LBA is relative to the start of the media. LBN is relative to the start of the volume.
- Volume: A set of sequentially consecutive blocks on the media allocated by the partitioning scheme. All blocks within a volume are between LBN 0 and LBN *n*, *n* being the last block in the volume. No sector or block outside of this area may be touched by the SFS file system driver.
- Boot Sector: The first block within the volume, LBN 0.

An Overlook of a Typical SFS Partition

There are a few items in a SFS volume that must be set to fixed values. The boot sector and location, size and location of the **Super Block**, and size and location of the **Index Data Area**.

The **Boot Sector** is always at **LBN** 0. Also residing in LBN 0 is a Super Block that contains information about the volume. At the end of the volume resides the Index Data Area. This Index Data Area holds information about the files stored on the volume. Some where in-between the Super Block and this Index Data Area is the Data Block Area as well as the Free Block Area.

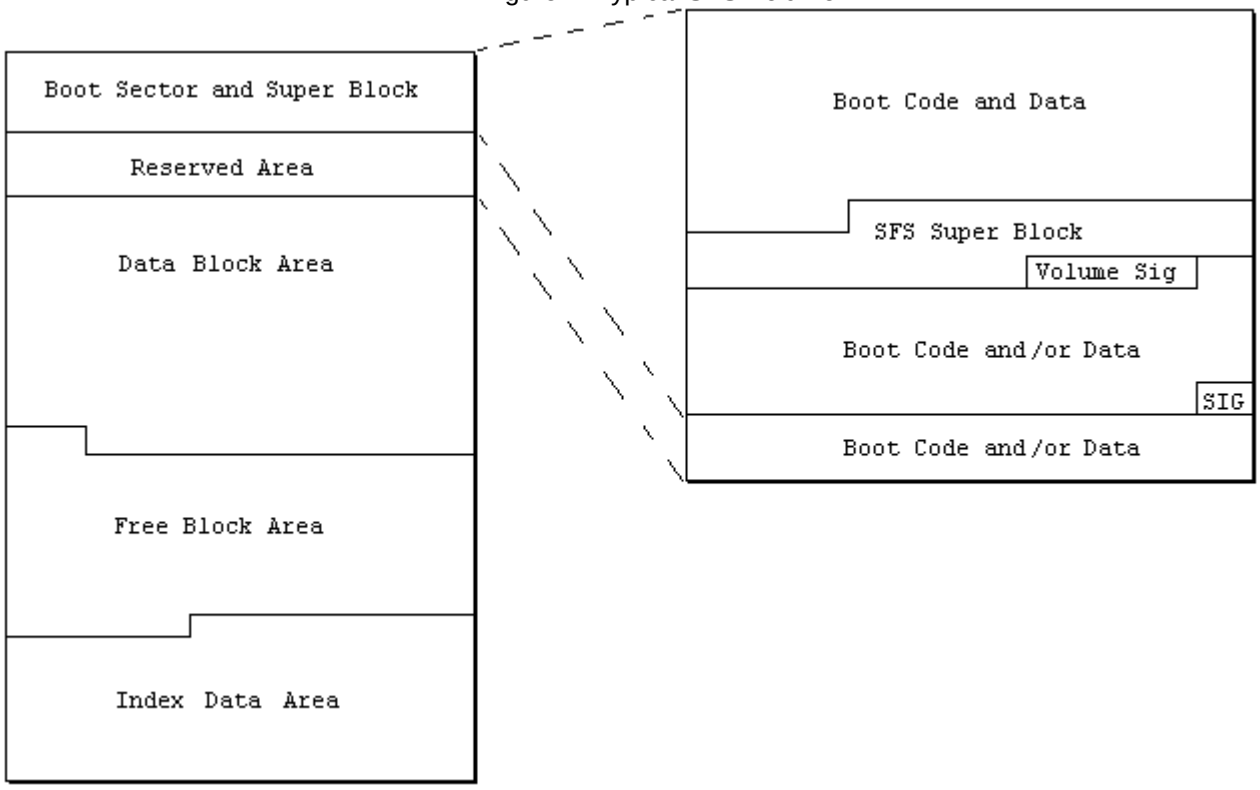
The SFS was originally designed to be compatible with other file systems in mind, therefore allowing a MBR with the standard four partition entries to reside in LBN 0 as well. However, this is a rare case and usually only would exist when no partitioning scheme is used. For example, a USB thumb drive can be formatted with a single volume occupying the whole media, with a FAT-32 file system. This would mean that a FAT-32 BIOS Parameter Block occupies the first block of the media. If the FAT-32 is formatted to only use the first half of the media, the SFS format could occupy the last part of the media.

This works because the SFS has its Super Block past the point of the FAT-32's BPB but before the point where any partition table entries would exist. Then the SFS marks the FAT-32 part of the media as reserved within its Super Block. See the original specification for more on this subject.

The remaining part of this document does not include, nor will it document any more information about how to have another file system co-exist in the same volume as the SFS. As far as this document is concerned, the SFS occupies the whole volume.

A typical SFS formatted volume will look something like what is shown in the figure below. The Boot Code and Data, Super Block, and any remaining boot code and/or data reside in LSB 0. Then any remaining sectors reserved for boot code or other data follow, with the Data Block Area, the Free Block Area, and finally the Index Data Area.

Figure 1: Typical SFS Volume



The Boot and Reserved Area

The first block contains any necessary code and data to either completely load a capable booting system or simply contains zeros when booting this volume is not required. If booting this volume is an option, the first block and any blocks in the Reserved Area are available for use. The size of the Reserved Area is predetermined and stored in the Super Block, which is stored within this first block of the volume and described next.

The Super Block

The Super Block is contained within the first block of the volume and has the format shown below. It must reside at **offset 0x18E** within this first block.

```
struct S_SFS_SUPER {
  bit64s time_stamp;    // Time Stamp of when volume has changed.
  bit64u data_size;     // Size of Data Area in blocks
  bit64u index_size;    // Size of Index Area in bytes
  bit8u  magic[3];      // signature   'SFS' (0x534653)
  bit8u  version;       // SFS version (0x10 = 1.0, 0x11 = 1.1)
  bit64u total_blocks;  // Total number of blocks in volume (including reserved area)
  bit32u rsvd_blocks;   // Number of reserved blocks
  bit8u  block_size;    // log(x+7) of block size (x = 2 = 512)
  bit8u  crc;           // Zero sum of bytes above
};
```

All LSB pointers in the Super Block are zero based from the start of the volume. All values are written and read in little-endian format, with least significant byte read/written first and most significant byte last.

The **time_stamp** field follows the format explained later and is updated anytime the data_size field or index_size field changes.

The **data_size** field is the count of blocks used for the Data Block Area. This is the area used to hold the file contents and explained later.

The **index_size** field is the size in bytes of the Index Data Area, the area holding the file metadata and is explained later.

The **magic** field is a three character signature field holding 'SFS' (0x534653) indicating that this is a Simple File System Super Block.

The **version** field holds the version of the specification this volume supports. The version is stored in BCD format, with the high nibble holding the major version and the low nibble holding the minor version.

The **total_blocks** field holds the 64-bit unsigned integer count of number of blocks in the whole volume. This is the count of all blocks from the block that holds the Super Block to the ending block that holds the last sector of the Index Data Area.

The **rsvd_blocks** field holds the 32-bit unsigned integer count of blocks before the Data Block Area. This includes the block holding the Super Block and any sequential blocks that may be present following this first block. This field is used to know where the Data Block Area begins. This field is relative to the start of the volume.

The **block_size** field is used to calculate the size of a block. This field is calculated as:

```
bytes per block = 2 ^ (block_size + 7)
```

For example, if the block size is 512, the standard size of most media, the block_size field will hold a value of 2. For 1,024-byte sectors, a value of 3 is used, and so on.

The **crc** field is a zero-byte check sum of the fields from the magic field to this field, inclusive, so that when each byte is added, ignoring any carryover larger than a byte, the sum will be zero.

The Data Block Area

The Data Block Area is where all of the file data is stored. Each file starts at a specified block indicated in the Index Data Area and is sequentially stored toward the end of the media, block by block. Each file starts on a block boundary.

There is no file metadata stored in this area, strictly only the contents of each file. All metadata is stored in the Index Data Area.

This area may grow or shrink depending on the usage of the volume. When a new file is to be added, the host may append to the end of the data area as long as there is enough sequentially spaced blocks to store the file data. Fragmentation of file data is prohibited. If the host deletes a file from the Index Data Area, the count of blocks used to store the file's data within this Data Block Area are said to be free for use. It is outside this specification on how the host is to maintain free blocks within this Data Block Area. No physical media within the SFS volume may be used to store or maintain this information.

Once a host finds that there are free blocks within the Data Block Area with enough sequentially stored blocks to hold the file's data, it may re-use these blocks without having to append to the Data Block Area.

If the host must append to the Data Block Area, it must first check that there is enough free space before the Index Data Area. Once it has found there is enough, it must update the data_size field in the Super Block to indicate the new size of the Data Block Area.

The Free Block Area

The Free Block Area is the blocks between the Data Block Area and the Index Data Area. These blocks are free for use to resize the Data Block Area, by adding blocks toward the end of the volume, or to resize the Index Data Area by adding blocks toward the start of the volume.

There are always free blocks in the Free Block Area until the point where the end of the Data Block Area meets the start of the Index Data Area. Once this happens, there are no more free blocks to use for file data unless the host has knowledge of unfragmented blocks within the Data Block Area. However, there is no more room for new unused entries in the Index Data Area.

The Index Data Area

The Index Data Area is where all of the file metadata is stored. This area holds a count of 64-byte entries holding various formats of data for the file system. The Index Data Area size is determined by the index_size field in the Super Block and must be a multiple of 64 bytes.

The first entry is at the end of the last block in the volume, with each sequential entry toward the start of the volume.

Please note that if an Index Block Entry contains continuation entries, explained later, the Index Data Entry associated with the continuation entries will be closest to the start of the volume, with each continuation entry after it, toward the end of the volume.

The first entry in the Index Data Area, remembering that this is the last 64 bytes of the last block of the volume, is the Volume ID entry. The last entry in the Index Data Area, remembering that this is the first 64 bytes of the Index Data Area is the Start Marker entry.

Note that if the size of the Index Data Area is not a multiple of the block size, the Start Marker entry may not be at the beginning of the block, but toward the end of the block depending on how many entries are in this first block.

There are eight (8) types of valid entries in the Index Data Area with one more type being a continuation entry. Each of the eight entries has a type field as the first byte of the 64-byte entry as well as a byte check sum. For example, the following shows the format of the Unused Entry type.

```
struct S_SFS_UNUSED {
  bit8u  type;        // 0x10
  bit8u  crc;         // zero sum byte check sum
  bit8u  resvd[62];   // reserved
};
```

Each type will start with this single unsigned 8-bit value ranging from 0x01 to 0x1A and shown below.

```
#define SFS_ENTRY_VOL_ID    0x01  // volume ID
#define SFS_ENTRY_START     0x02  // start marker
#define SFS_ENTRY_UNUSED    0x10  // unused
#define SFS_ENTRY_DIR       0x11  // directory entry
#define SFS_ENTRY_FILE      0x12  // file entry
#define SFS_ENTRY_UNUSABLE  0x18  // unusable entry (bad sector(s))
#define SFS_ENTRY_DIR_DEL   0x19  // deleted directory
#define SFS_ENTRY_FILE_DEL  0x1A  // deleted file
```

The **type** field must be one of the eight listed above. If it is any other value, it is either an invalid entry or part of a continuation entry, explained later.

The **crc** field is the byte sum of all 64 bytes in this entry and all 64 bytes of each continuation entry linked to this entry.

Please note that even if not all of the bytes are used in the continuation entries for that particular entry type, all 64 bytes are still calculated within this crc value and this crc field only exists in the first entry before any continuation entries.

The Volume ID Entry

The Volume ID entry is as follows and holds the name of this volume as well as a time stamp of when it was created. All SFS volumes must contain this entry and it must be the first entry in the Index Data Area

```
struct S_SFS_VOL_ID {
  bit8u  type;        // 0x01
  bit8u  crc;         // zero sum byte check sum
  bit16u resvd;       // reserved
  bit64s time_stamp;  // time of media format/volume creation
  bit8u  name[52];    // UTF-8 null terminated
};
```

All reserved fields in any entry type are to be written as zeros at creation time and preserved when read and written again.

The **time_stamp** field is written to at volume creation time and is then not normally ever changed. This field holds the time of day when the volume was created and this format is explained later.

The **name** field holds the volume name usually written to at creation time but may change throughout the life of the volume. The format of this field and all character-based fields is explained later.

The Start Marker Entry

The Start Marker entry is as follows and marks the last entry in the Index Data Area.

```
struct S_SFS_START {
  bit8u  type;        // 0x02
  bit8u  crc;         // zero sum byte check sum
  bit8u  resvd[62];   // reserved
};
```

Remember that the Start Marker Entry is the entry closest to the start of the volume and may or may not be on a block boundary. For example, if the Block Size is 512 bytes and the Index Data Area is 1,408 bytes, the Index Data Area will occupy the last three blocks of the volume. However, the first 128 bytes of the first block, the block closest to the start of the volume, will be unused, with the Start Marker Entry at offset 128 within the block.

Since the Free Block Area cannot use partial blocks, it is usually common for the Index Data Area to use all of the block closest to the start of the volume and have the Start Marker Entry at the start of that block, adding Unused entries as needed. Note that this not a requirement and must not be assumed to be. Any space before the Start Marker Entry, relative to the start of that block, is considered unused, undefined, and may not be assumed to have any set value.

The Unused Entry

The Unused entry is as follows and is simply an available entry for use as a different type when needed.

```
struct S_SFS_UNUSED {
  bit8u  type;        // 0x10
  bit8u  crc;         // zero sum byte check sum
  bit8u  resvd[62];   // reserved
};
```

The Directory Entry

The Directory Entry is as follows and simply holds the name of a directory contained on the volume.

```
#define DIR_NAME_LEN  53
struct S_SFS_DIR {
  bit8u  type;        // 0x11
  bit8u  crc;         // zero sum byte check sum
  bit8u  num_cont;    // number of cont slots
  bit64s time_stamp;  //
  bit8u  name[DIR_NAME_LEN]; // UTF-8 null terminated
};
```

This entry only holds a name for a directory existing on the volume. No other information is given since the full path for each file is placed within its respected File Entry.

The **num_cont** field is an 8-bit unsigned integer count of number of continuation entries that follow this entry used to store the name and path if the name field is not large enough to do so. Continuation entries are stored sequentially after this entry and area explained later.

The **time_stamp** field may be updated by the host when the directory is created or any file or directory within it is also created, moved, renamed, or deleted. It is not part of this specification on when or how often this field is updated.

The **name** field holds all or the first part of the full path of this directory using continuation entries as needed.

The File Entry

The File Entry is as follows and holds all of the metadata needed to store the file on the volume.

```
#define FILE_NAME_LEN  29
struct S_SFS_FILE {
  bit8u  type;        // 0x12
  bit8u  crc;         // zero sum byte check sum
  bit8u  num_cont;    // number of cont slots
```

```
  bit64s time_stamp;  //
  bit64u start_block; // starting block in data area
  bit64u end_block;   // end block in data area
  bit64u file_len;    // file length in bytes
  bit8u  name[FILE_NAME_LEN]; // UTF-8 null terminated
};
```

This entry holds the necessary data to find and read the data contents for this file.

The **num_cont** field is an 8-bit unsigned integer count of number of continuation entries that follow this entry used to store the name and path if the name field is not large enough to do so. Continuation entries are stored sequentially after this entry and area explained later.

The **time_stamp** field is updated by the host when this file is created or modified.

The **start_block** and **end_block** entries indicate the starting block and ending block of the file's contents relative to the start of the volume, the same block that stores the Super Block. If the file does not contain any data, i.e.: the file is a zero length file, then no blocks within the Data Block Area are used and both these entries are written zero.

The **name** field holds all or the first part of the full path of this filename using continuation entries as needed.

The Unusable Entry

The Unusable Entry is as follows and holds a starting and ending block number of unusable blocks.

```
struct S_SFS_UNUSABLE {
  bit8u  type;        // 0x18
  bit8u  crc;         // zero sum byte check sum
  bit8u  resv0[8];    // reserved
  bit64u start_block; // starting block in data area
  bit64u end_block;   // end block in data area
  bit8u  resv1[38];   // reserved
};
```

Unusable blocks are blocks that are not usable by the host. For example, there could be bad sectors on the media and this entry is used to indicate which ones. If there are more than one area, non-sequentially linked, you must have one entry for each non-sequential set of unusable sectors.

The Deleted Directory and File Entries

The Deleted Directory and Deleted File Entries are as follows and are identical to their respective entries except that the type field has changed.

```
struct S_SFS_DIR_DEL {
  bit8u  type;        // 0x19
  bit8u  crc;         // zero sum byte check sum
  bit8u  num_cont;    // number of cont slots
  bit64s time_stamp;  //
  bit8u  name[DIR_NAME_LEN]; // UTF-8 null terminated
};

struct S_SFS_FILE_DEL {
  bit8u  type;        // 0x1A
  bit8u  crc;         // zero sum byte check sum
  bit8u  num_cont;    // number of cont slots
  bit64s time_stamp;  //
  bit64u start_block; // starting block in data area
```

```
  bit64u end_block;   // end block in data area
  bit64u file_len;    // file length in bytes
  bit8u  name[FILE_NAME_LEN]; // UTF-8 null terminated
};
```

This is so that you may undelete a file or directory if you so desire. This is where the crc field comes into play. You must verify that all bytes within this entry and any continuation entries associated with this entry pass the byte-sum check before you undelete a directory or file name.

For example, the host system might see a deleted entry as an unused entry and each continuation entry associated with it as also unused. It then might reassign one of the continuation entries but not the deleted entry.

Note that if you do not wish to have Undelete capabilities, mark all deleted entries, including the associated continuation entries, as Unused entries. It is not good practice to reassign "deleted" continuation entries without reassigning the deleted parent entry as well.

Note that just because the Deleted Entry is still valid, this doesn't mean that the blocks used to store the deleted file's data are still intact. It is up to the host to determine if they are still intact and not part of this specification.

Filenames and Directories

As other file systems may store directory information within the media's data area, the SFS does not. Each file has its full path name stored within its File Entry. Therefore, there is no need to store directory information elsewhere. However, a Directory Entry is included to simply indicate the name and existence of that directory. It also contains its full path and name.

For example, a file that may be stored in the (virtual) root directory will simply hold the name of the file. If the volume includes a directory name, "foobar" for example, there will be a Directory Entry with that name. Then any file within the directory will have that name followed by a forward slash and then the name of the file as in "foobar/name.txt".

Note that there is no forward slash at the beginning of the path.

All names, File, Directory, or Volume IDs, are stored as UTF-8 character strings and are null terminated. All character codes less than `0x0020`, codes `0x0080` through `0x009F` inclusively, and all of the following codes are invalid codes and should not be used in any name.

- `"` (double quote, `0x0022`)
- `*` (asterix, `0x002A`)
- `:` (colon, `0x003A`)
- `<` (less than sign, `0x003C`)
- `>` (greater than sign, `0x003E`)
- `?` (question mark, `0x003F`)
- `\` (backward slash, `0x005C`)
- DEL (delete, `0x007F`)
- NBSP (no break space character, `0x00A0`)

This includes forward slashes in file names since forward slashes indicate a path separation.

Continuation Entries

Since the space allocated within a given entry may not be enough to hold the full path and file name, the SFS specification gives the ability to allocate more space by assigning sequentially consecutive entries as continuation

entries. These entries must follow the associated entry in the direction toward the end of the volume. The associated entry will indicate how many continuation entries are used.

A continuation entry does not have a type field or any other structured data within it. A continuation entry is just a 64-byte entry to continue the name from the previous entry. With an 8-bit unsigned integer, this allows up to 255 continuation entries to each associated named entry giving a path and name length limit of 16,320 bytes not including the space within the associated entry.

Note that the continuation entries will be sequentially consecutive in memory, in turn simply turning the short 53-byte directory name field into a 16,373-byte field.

If a continuation entry is not needed, the num_cont field in the associated entry will be zero.

Remember that all name strings must be null terminated, meaning that all strings must end in a 0x0000 code. If the complete path will fit within the room allocated in the associated entry, but is not null terminated, you must still allocate a continuation entry just for the null terminator.

Note that most ASCII characters are UTF-8 characters and will occupy a single byte per character.

Time Stamp

Some entries and the Super Block contain a 64-bit time_stamp field. This field is the signed integer count of 1/65536^{ths} of a second before or after an EPOC time of midnight, January 1st, 1970. With this in mind, a value of one (1) is roughly 15 micoseconds, with a value of 3,932,160 being one minute past midnight.

This is a much more accurate or precise measurement of time. Usually, with a write-once file system, this is not an issue, but the author decided to use this anyway. Precise time stamps are quite valuable in MAKE instances when a file's time stamp indicates if it needs to be built or not.

Changes

This specification adds function and features to the existing version 1.0 of the specification, which now breaks that version. If you plan on supporting version 1.0, please make sure and check the version value in the Super Block.

Version 1.1 (breaks version 1.0)

- Modifications and additions by Benjamin David Lunt.
- Moved the Super Block six bytes toward the start of the media to leave room for the common signature area.
- Added the CRC byte field to each entry. This moves each following field by an offset of one as well as decrements the length of the allocated space for the names.
- Changed the not-used indicator for not-used blocks from both being zero to both being -1. If both are zero, this still could mean that it uses the first and only the first block of the Data Block Area. It is more likely that there will not be 2^64-1 blocks on the media, hence the -1 indicator.
- Version 1.0 specified that an Ending Block must be higher than a Starting Block. This is not the case in this version. The Ending Block must be equal to or higher than the Starting Block.

Version 1.0

- Initial release by Brendan Trotter

Examples

The rest of this specification is a list of examples and DEBUG style dumps to show exactly what a volume might look like.

The first example is a Super Block from a 1.44Meg floppy disk.

Super Block:

```
00000180                                                 24 86                  ..
00000190  52 E1 B9 59 00 00 23 0A-00 00 00 00 00 00 80 04     R..Y..#.........
000001A0  00 00 00 00 00 00 53 46-53 11 40 0B 00 00 00 00     ......SFS.@.....
000001B0  00 00 02 00 00 00 02 B4                             ........
```

offset	contents	description
0000018E	0x000059B9E1528624	Time Stamp
00000196	0x0000000000000A23	Data Block Count (2,595 blocks used)
0000019E	0x0000000000000480	Index Size in bytes (1,152 bytes)
000001A6	0x534653	Magic Signature ‘SFS’
000001A9	0x11	Version 1.1
000001AA	0x0000000000000B40	Total Blocks in Volume (2,880 blocks)
000001B2	0x00000002	Reserved Blocks (2)
000001B6	0x02	Block Size (2 = 512)
000001B7	0xB4	Byte Sum CRC

Second is an example of a File Entry that does not need a continuation.

File Entry: (64 bytes)

```
00167C00  12 63 00 24 86 52 E1 B9-59 00 00 49 03 00 00 00     .c...R..Y..I....
00167C10  00 00 00 F2 03 00 00 00-00 00 00 B0 50 01 00 00     ............P...
00167C20  00 00 00 6C 6F 61 64 65-72 2E 73 79 73 00 00 00     ...loader.sys...
00167C30  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
```

offset	contents	description
00167C00	0x12	File Entry Type
00167C01	0x63	Byte Sum CRC*
00167C02	0x00	No Continuation Entries used
00167C03	0x000059B9E1528624	Time Stamp
00167C0B	0x0000000000000349	Start Block (relative to Data Block Area)
00167C13	0x00000000000003F2	End Block (relative to Data Block Area)
00167C1B	0x00000000000150B0	File Length in bytes (86,192 bytes)
00167C23	‘loader.sys’,0,...	Null terminated name

Now for an entry that uses name continuation.

File Entry: (64 bytes)

```
00167C40  12 B4 01 24 86 52 E1 B9-59 00 00 F3 03 00 00 00     .....R..Y..I....
00167C50  00 00 00 F4 03 00 00 00-00 00 00 75 10 00 00 00     ................
00167C60  00 00 00 61 20 6C 6F 6E-67 20 73 74 69 6E 6B 27     ...a.long.stink'
00167C70  6E 20 6E 61 6D 65 20 74-68 61 74 20 6A 75 73 74     n.name.that.just
00167C80  20 67 6F 65 73 20 6F 6E-20 61 6E 64 20 6F 6E 00     .goes.on.and.on.
00167C90  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00167CA0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00167CB0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
```

offset	contents	description
00167C40	0x12	File Entry Type
00167C41	0xB4	Byte Sum CRC*
00167C42	0x01	One Continuation Entries used
00167C43	0x000059B9E1528624	Time Stamp
00167C4B	0x00000000000003F3	Start Block (relative to Data Block Area)

```
00167C13  0x00000000000003F4  End Block (relative to Data Block Area)
00167C1B  0x0000000000000275  File Length in bytes (629 bytes)
00167C23  'a long stink'n name that just goes on and on',0..
```

*Note that the CRC byte shown above might not be correct due to modification of the examples while writing this documentation.

Wrap Up

Remember that each File and Directory entry contains the full path within its entry. For example, consider the following directory tree:

```
\
  foo\
  foo\bar.txt
  foo\bar.bin
  bar\
  bar\foo.txt
  bar\foo.bin
```

If this is the only thing on the volume, not counting the Start Marker Entry and the Volume ID Entry, you will have six (6) total entries, two (2) Directory Entries and four (4) File Entries. The File Entry for the last file in the list above, "bar\foo.bin" will have its name entry contain:

```
'bar/foo.bin',0
```

Minus the single quotes and comma of course.

FYSOS Specific

The FYSOS project has specific specifications for bootable volumes. For example, the boot code must pass certain information to the loader, which in turn passes this and more information to the kernel. One of these items is the base block address of this volume along with a signature unique to this volume.

The kernel itself does not need the base address, but the boot loader does. For example, the partitioning scheme was able to gather the base address of the first sector of your boot code. However, since each partitioning scheme is different, how does the boot code know where it is and where to load any remaining sectors? Some file systems account for this with Reserved Sector fields or other indicators. However, some file systems do not, specifically the one explained in this document.

Therefore, all FYSOS boot sector code reserves the 12 bytes just before the 0xAA55 signature in the first sector to indicate the base address and this unique signature. This structure is located at offset 0x1F2 within the first sector of the volume no matter the size of the sector or block used.

```
struct S_FYSOS_BOOT_SIG {
  bit32u signature;   // unique signature to this volume
  bit64u base_addr;   // base LBA of this sector/block
  bit16u boot_sig;    // BIOS expected boot signature of 0xAA55 (0x55 0xAA)
};
```

These 12 bytes will remain reserved for use by the FYSOS boot sector specification for all FYSOS bootable volumes, and will be assigned and written to at volume build time.