The MSF File Format

File Layout

The MSF file format consists of the following components:

1. The Superblock
2. The Free Block Map (also know as Free Page Map, or FPM)
3. Data

Each component is stored as an indexed block, the length of which is specified in `SuperBlock::BlockSize`. The file consists of 1 or more iterations of the following pattern (sometimes referred to as an "interval"):

1. 1 block of data
2. Free Block Map 1 (corresponds to `SuperBlock::FreeBlockMapBlock` 1)
3. Free Block Map 2 (corresponds to `SuperBlock::FreeBlockMapBlock` 2)
4. `SuperBlock::BlockSize - 3` blocks of data

In the first interval, the first data block is used to store The Superblock.

The following diagram demonstrates the general layout of the file (| denotes the end of an interval, and is for visualization purposes only):

Block Index	**0**	**1**	**2**	**3 – 4095**		**4096**	**4097**	**4098**	**4099 – 8191**		**...**
Meaning	The Superblock	Free Block Map 1	Free Block Map 2	Data	|	Data	FPM1	FPM2	Data	|	...

The file may end after any block, including immediately after a FPM1.

Note

LLVM only supports 4096 byte blocks (sometimes referred to as the "BigMsf" variant), so the rest of this document will assume a block size of 4096.

The Superblock

At file offset 0 in an MSF file is the MSF *SuperBlock*, which is laid out as follows:

```
struct SuperBlock {
  char FileMagic[sizeof(Magic)];
  ulittle32_t BlockSize;
  ulittle32_t FreeBlockMapBlock;
  ulittle32_t NumBlocks;
  ulittle32_t NumDirectoryBytes;
  ulittle32_t Unknown;
  ulittle32_t BlockMapAddr;
};
```

- **FileMagic** – Must be equal to `"Microsoft C / C++ MSF 7.00\\r\\n"` followed by the bytes `1A 44 53 00 00 00`.
- **BlockSize** – The block size of the internal file system. Valid values are 512, 1024, 2048, and 4096 bytes. Certain aspects of the MSF file layout vary depending on the block sizes. For the purposes of LLVM, we handle only block sizes of 4KiB, and all further discussion assumes a block size of 4KiB.
- **FreeBlockMapBlock** – The index of a block within the file, at which begins a bitfield representing the set of all blocks within the file which are "free" (i.e. the data within that block is not used). See The Free Block Map for more information. **Important**: `FreeBlockMapBlock` can only be `1` or `2`!
- **NumBlocks** – The total number of blocks in the file. `NumBlocks * BlockSize` should equal the size of the file on disk.
- **NumDirectoryBytes** – The size of the stream directory, in bytes. The stream directory contains information about each stream's size and the set of blocks that it occupies. It will be described in more detail later.
- **BlockMapAddr** – The index of a block within the MSF file. At this block is an array of `ulittle32_t`'s listing the blocks that the stream directory resides on. For large MSF files, the stream directory (which describes the block layout of each stream) may not fit entirely on a single block. As a result, this extra layer of indirection is introduced, whereby this block contains the list of blocks that the stream directory occupies, and the stream directory itself can be stitched together accordingly. The number of `ulittle32_t`'s in this array is given by `ceil(NumDirectoryBytes / BlockSize)`.

The Free Block Map

The Free Block Map (sometimes referred to as the Free Page Map, or FPM) is a series of blocks which contains a bit flag for every block in the file. The flag will be set to 0 if the block is in use, and 1 if the block is unused.

Each file contains two FPMs, one of which is active at any given time. This feature is designed to support incremental and atomic updates of the underlying MSF file. While writing to an MSF file, if the active FPM is FPM1, you can write your new modified bitfield to FPM2, and vice versa. Only when you commit the file to disk do you need to swap the value in the SuperBlock to point to the new `FreeBlockMapBlock`.

The Free Block Maps are stored as a series of single blocks throughout the file at intervals of BlockSize. Because each FPM block is of size `BlockSize` bytes, it contains 8 times as many bits as an interval has blocks. This means that the first block of each FPM refers to the first 8 intervals of the file (the first 32768 blocks), the second block of each FPM refers to the next 8 blocks, and so on. This results in far more FPM blocks being present than are required, but in order to maintain backwards compatibility the format must stay this way.

The Stream Directory

The Stream Directory is the root of all access to the other streams in an MSF file. Beginning at byte 0 of the stream directory is the following structure:

```
struct StreamDirectory {
  ulittle32_t NumStreams;
  ulittle32_t StreamSizes[NumStreams];
  ulittle32_t StreamBlocks[NumStreams][];
};
```

And this structure occupies exactly `SuperBlock->NumDirectoryBytes` bytes. Note that each of the last two arrays is of variable length, and in particular that the second array is jagged.

Example: Suppose a hypothetical PDB file with a 4KiB block size, and 4 streams of lengths {1000 bytes, 8000 bytes, 16000 bytes, 9000 bytes}.

Stream 0: ceil(1000 / 4096) = 1 block

Stream 1: ceil(8000 / 4096) = 2 blocks

Stream 2: ceil(16000 / 4096) = 4 blocks

Stream 3: ceil(9000 / 4096) = 3 blocks

In total, 10 blocks are used. Let's see what the stream directory might look like:

```
struct StreamDirectory {
  ulittle32_t NumStreams = 4;
  ulittle32_t StreamSizes[] = {1000, 8000, 16000, 9000};
  ulittle32_t StreamBlocks[][] = {
    {4},
    {5, 6},
    {11, 9, 7, 8},
    {10, 15, 12}
  };
};
```

In total, this occupies `15 * 4 = 60` bytes, so `SuperBlock->NumDirectoryBytes` would equal `60`, and `SuperBlock->BlockMapAddr` would be an array of one `ulittle32_t`, since `60 <= SuperBlock->BlockSize`.

Note also that the streams are discontiguous, and that part of stream 3 is in the middle of part of stream 2. You cannot assume anything about the layout of the blocks!

Alignment and Block Boundaries

As may be clear by now, it is possible for a single field (whether it be a high level record, a long string field, or even a single `uint16`) to begin and end in separate blocks. For example, if the block size is 4096 bytes, and a `uint16` field begins at the last byte of the current block, then it would need to end on the first byte of the next block. Since blocks are not necessarily contiguously laid out in the file, this means that both the consumer and the producer of an MSF file must be prepared to split data apart accordingly. In the aforementioned example, the high byte of the `uint16` would be written to the last byte of block N, and the low byte would be written to the first byte of block N+1, which could be tens of thousands of bytes later (or even earlier!) in the file, depending on what the stream directory says.