AAF ASSOCIATION SPECIFICATION

Advanced Authoring Format (AAF) Low-Level Container Specification v1.0.1

Compound File Binary File Format

A compound file, also known as a structured storage file, is made up of a number of virtual streams. These are collections of data that behave as a linear stream, although their on-disk format may be fragmented. Virtual streams can contain user data, or they can contain control structures used to maintain the file. The file itself can also be considered a virtual stream.

All allocations of space within a compound file are done in units called sectors. The size of a sector is definable at creation time of a compound file, but for the purposes of this document it is always either 512 bytes or 4 kilobytes (KB). A virtual stream is made up of a sequence of sectors.

The compound file uses several different types of sectors: file allocation table (FAT), directory, mini file allocation table (MiniFAT), double-indirect FAT (DIFAT), Stream, and Range Lock. A header is a separate type of sector, which is always located at offset zero. If the compound file is greater than 2 gigabytes (GB) in size, a special range lock sector is located at offset 2 GB. With the exception of the header and the range lock sector, sectors of any type can be placed anywhere within the file. The function of the various sector types is discussed below.

In the discussion below, the term SECT is used to describe the location of a sector within a virtual stream (in most cases this virtual stream is the file itself). Internally, a SECT is represented as a 32-bit ULONG.

Sector Types

The following C language data type definitions describe the length of the various fields in each on-disk structure.

```
typedef unsigned long ULONG;    // 4 bytes
typedef unsigned short USHORT;  // 2 bytes
typedef short OFFSET;           // 2 bytes
typedef ULONG SECT;             // 4 bytes
typedef ULONG FSINDEX;          // 4 bytes
typedef USHORT FSOFFSET;        // 2 bytes
typedef USHORT WCHAR;           // 2 bytes
typedef ULONG DFSIGNATURE;      // 4 bytes
typedef unsigned char BYTE;     // 1 byte
typedef unsigned short WORD;    // 2 bytes
typedef unsigned long DWORD;    // 4 bytes
typedef ULONG SID;              // 4 bytes
typedef GUID CLSID;             // 16 bytes

// 64-bit value representing number of 100 nanoseconds since January 1, 1601
typedef struct tagFILETIME {
  DWORD dwLowDateTime;
  DWORD dwHighDateTime;
} FILETIME;
```

```
const SECT MAXREGSECT = 0xFFFFFFFA; // maximum SECT
const SECT DIFSECT    = 0xFFFFFFFC; // denotes a DIFAT sector in a FAT
const SECT FATSECT    = 0xFFFFFFFD; // denotes a FAT sector in a FAT
const SECT ENDOFCHAIN = 0xFFFFFFFE; // end of a virtual stream chain
const SECT FREESECT   = 0xFFFFFFFF; // unallocated sector

const SID MAXREGSID   = 0xFFFFFFFA; // maximum directory entry ID
const SID NOSTREAM    = 0xFFFFFFFF; // unallocated directory entry
```


Header

```
struct StructuredStorageHeader { // [offset from start (bytes), length (bytes)]
  BYTE _abSig[8];            // [00H,08] {0xd0, 0xcf, 0x11, 0xe0, 0xa1, 0xb1,
                             //   0x1a, 0xe1} for current version
  CLSID _clsid;              // [08H,16] reserved must be zero (WriteClassStg/
                             //   GetClassFile uses root directory class id)
  USHORT _uMinorVersion;     // [18H,02] minor version of the format: 33 is
                             //   written by reference implementation
  USHORT _uDllVersion;       // [1AH,02] major version of the dll/format: 3 for
                             //   512-byte sectors, 4 for 4 KB sectors
  USHORT _uByteOrder;        // [1CH,02] 0xFFFE: indicates Intel byte-ordering
  USHORT _uSectorShift;      // [1EH,02] size of sectors in power-of-two;
                             //   typically 9 indicating 512-byte sectors
  USHORT _uMiniSectorShift;  // [20H,02] size of mini-sectors in power-of-two;
                             //   typically 6 indicating 64-byte mini-sectors
  USHORT _usReserved;        // [22H,02] reserved, must be zero
  ULONG _ulReserved1;        // [24H,04] reserved, must be zero
  FSINDEX _csectDir;         // [28H,04] must be zero for 512-byte sectors,
                             //   number of SECTs in directory chain for 4 KB
                             //   sectors
  FSINDEX _csectFat;         // [2CH,04] number of SECTs in the FAT chain
  SECT _sectDirStart;        // [30H,04] first SECT in the directory chain
  DFSIGNATURE _signature;    // [34H,04] signature used for transactions; must
                             //   be zero. The reference implementation
                             //   does not support transactions
  ULONG _ulMiniSectorCutoff; // [38H,04] maximum size for a mini stream;
                             //   typically 4096 bytes
  SECT _sectMiniFatStart;    // [3CH,04] first SECT in the MiniFAT chain
  FSINDEX _csectMiniFat;     // [40H,04] number of SECTs in the MiniFAT chain
  SECT _sectDifStart;        // [44H,04] first SECT in the DIFAT chain
  FSINDEX _csectDif;         // [48H,04] number of SECTs in the DIFAT chain
  SECT _sectFat[109];        // [4CH,436] the SECTs of first 109 FAT sectors
};
```

The header contains vital information for instantiating a compound file. Its total length is 512 bytes. There is exactly one header in any compound file, and it is always located beginning at offset zero in the file. If the sector size is greater than 512 bytes, then the header is padded to the sector size with zeroes.

In Microsoft® Windows® 2000 and later, a new major version of the compound file binary format, 4, was added to support a sector size of 4096 bytes. When creating 4 KB sector compound files, set _uDllVersion to 4, _uSectorShift to 12 (corresponds to 4 KB), and _csectDir to the number of directory sectors.

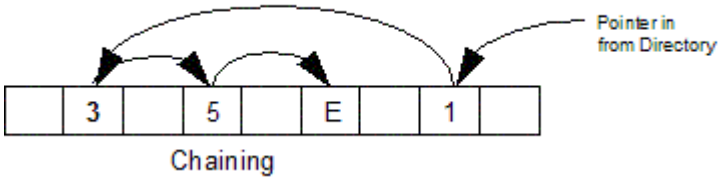
To allow maximum file portability, the _uByteOrder field should be set to 0xFFFE, which indicates little-endian byte ordering. Implementations with big-endian architectures (such as Macintosh) should byte-swap all integers when

reading or writing the any on-disk structure.

FAT Sectors

The FAT is the main allocator for space within a compound file. Every sector in the file is represented within the FAT in some fashion, including those sectors that are unallocated (free). The FAT is a virtual stream made up of one or more FAT sectors.

FAT sectors are arrays of SECTs that represent the allocation of space within the file. Each stream is represented in the FAT by a chain, in much the same fashion as a disk operating system (DOS) FAT The set of FAT sectors can be considered together as a single array—each cell in that array contains the SECT of the next sector in the chain, and this SECT can be used as an index into the FAT array to continue along the chain. Special values are reserved for chain terminators (ENDOFCHAIN = 0xFFFFFFFE), free sectors (FREESECT = 0xFFFFFFFF), and sectors that contain storage for FAT sectors (FATSECT = 0xFFFFFFFD) or DIFAT sectors (DIFSECT = 0xFFFFFFC), which are not chained in the same way as the others.



The locations of FAT sectors are read from the DIFAT, which is described below. The FAT is represented in itself, but not by a chain—a special reserved SECT value (FATSECT = 0xFFFFFFFD) is used to mark sectors allocated to the FAT.

A SECT can be converted into a byte offset into the file by using the following formula: (SECT+1) << _uSectorShift. This implies that sector 0 of the file begins at byte offset 1 << _uSectorShift, not at 0.

In the example above, the virtual stream chain starts at sector #7, continues to sector #1, continues to sector #3, and ends with sector #5. The next sector for sector #5 points to ENDOFCHAIN.

MiniFAT Sectors

Since space for streams is always allocated in sector-sized blocks, there can be considerable waste when storing objects much smaller than sectors (typically 512 bytes). As a solution to this problem, Microsoft introduced the concept of the MiniFAT. The MiniFAT is structurally equivalent to the FAT, but is used in a different way. The virtual sector size for objects represented in MiniFAT is 1 << _uMiniSectorShift (typically 64 bytes) instead of 1 << _uSectorShift (typically 512 bytes or 4 KB). The storage for these objects comes from a virtual stream called the mini stream.

The locations for MiniFAT sectors are stored in a standard chain in the FAT, with the beginning of the chain stored in the header (_sectMiniFatStart).

A MiniFAT sector number can be converted into a byte offset into the mini stream by using the following formula: SECT << _uMiniSectorShift. This formula is different from the formula used to convert a SECT into a byte offset in the file, since no header is stored in the mini stream.

The mini stream is chained within the FAT in exactly the same fashion as any normal stream. The mini stream's starting sector is referenced in the first directory entry (SID 0).

If all of the user streams in the file are greater than the cutoff (typically 4 KB for _ulMiniSectorCutoff), then MiniFAT and mini stream are not required. In this case, _sectMiniFatStart can be set to ENDOFCHAIN, and the root directory entry's _sectStart can be set to NOSTREAM.

DIFAT Sectors

The DIFAT is used to represent storage of the FAT. The DIFAT is also represented by an array of SECTs, and is chained by the terminating cell in each sector array. As an optimization, the first 109 FAT sectors are represented

within the header itself, so no DIFAT sectors will be found in a small compound file. A compound file is considered small if it is smaller than 7 megabyte (MB) for 512 byte sectors.

The DIFAT represents the FAT in a different manner than the FAT represents a chain. A given index into the DIFAT will contain the SECT of the FAT sector found at that offset in the FAT virtual stream. For instance, index 3 in the DIFAT contains the SECT for sector #3 of the FAT.

The storage for DIFAT sectors is reserved in the FAT, but is not chained there. Space for DIFAT sectors is reserved by a special SECT value, DIFSECT=0xFFFFFFFC. The location of the first DIFAT sector is stored in the header.

A special value of ENDOFCHAIN=0xFFFFFFFE is stored in the pointer to the next DIFAT sector of the last DIFAT sector.

Directory Sectors

```
typedef enum tagSTGTY {
  STGTY_INVALID     = 0,       // unknown storage type
  STGTY_STORAGE     = 1,       // element is a storage object
  STGTY_STREAM      = 2,       // element is a stream object
  STGTY_LOCKBYTES   = 3,       // element is an ILockBytes object
  STGTY_PROPERTY    = 4,       // element is an IPropertyStorage object
  STGTY_ROOT        = 5        // element is a root storage
} STGTY;

typedef enum tagDECOLOR {
  DE_RED            = 0,
  DE_BLACK          = 1
} DECOLOR;

// [offset from start (bytes), length (bytes)]
struct StructuredStorageDirectoryEntry {
  WCHAR _ab[32];      // [00H,64] 64 bytes. The element name in Unicode, padded
                      //   with zeros to fill this byte array.  Terminating
                      //   Unicode NULL is required.
  WORD _cb;           // [40H,02] Length of the Element name in bytes,
                      //   including the Unicode NULL
  BYTE _mse;          // [42H,01] Type of object. Value taken from the
                      //   STGTY enumeration
  BYTE _bflags;       // [43H,01] Value taken from DECOLOR enumeration
  SID _sidLeftSib;    // [44H,04] SID of the left-sibling of this entry
                      //   in the directory tree
  SID _sidRightSib;   // [48H,04] SID of the right-sibling of this entry
                      //   in the directory tree
  SID _sidChild;      // [4CH,04] SID of the child acting as the root of all
                      //   the children of this element
                      //   (if _mse=STGTY_STORAGE or STGTY_ROOT)
  CLSID _clsId;       // [50H,16] CLSID of this storage
                      //   (if _mse=STGTY_STORAGE or STGTY_ROOT)
  DWORD _dwUserFlags; // [60H,04] User flags of this storage
                      //   (if _mse=STGTY_STORAGE or STGTY_ROOT)
  FILETIME _time[2];  // [64H,16] Create/Modify time-stamps
                      //   (if _mse=STGTY_STORAGE)
  SECT _sectStart;    // [74H,04] starting SECT of the stream
                      //   (if _mse=STGTY_STREAM)
  ULONG _ulSizeLow;   // [78H,04] size of stream in bytes
                      //   (if _mse=STGTY_STREAM)
  ULONG _ulSizeHigh;  // [7CH,02] must be zero for 512-byte sectors,
                      //   high part of 64-bit size for 4 KB sectors
};
```

The directory is a structure used to contain per-stream information about the streams in a compound file, as well as to maintain a tree-style containment structure. It is a virtual stream made up of one or more directory sectors. The directory is represented as a standard chain of sectors within the FAT. The first sector of the directory chain is the root directory entry.

Each level of the containment hierarchy (that is, each set of siblings) is represented as a red/black tree. The parent of this set of siblings will have a pointer to the top of this tree. This red/black tree must maintain the following

conditions in order for it to be valid:

1. The root node must always be black. Since the root directory (see below) does not have siblings, its color is irrelevant and may therefore be either red or black.
2. No two consecutive nodes may both be red.
3. The left sibling must always be less than the right sibling. This relationship is defined as:
    - A node with a shorter name is less than a node with a longer name (compare the length of the name)
    - For nodes with the same length names, compare the two names with a case-insensitive alphabetic sort.

The simplest implementation of the above invariants would be to mark every node as black, in which case the tree is simply a binary tree.

A directory sector is an array of directory entries. Each user stream within a compound file is represented by a single directory entry. The directory is considered as a large array of directory entries. It is useful to note that the directory entry for a stream remains at the same index in the directory array for the life of the stream—thus, this index, called a stream identifier (SID) can be used to readily identify a given stream. A special SID (called NOSTREAM) is used to denote an empty pointer. A compound file SID is not the same as a Windows security identifier (SID), which is a security identifier for user accounts.

The directory entry is then padded out with zeros to make a total size of 128 bytes. The name in the directory entry is limited to 32 Unicode characters, including the required Unicode null terminator.

Directory entries are grouped into blocks to form directory sectors. There are 4 directory entries in a 512-byte directory sector, and there are 32 directory entries in a 4 KB directory sector.

Root Directory Entry

The first sector of the directory chain (also referred to as the first element of the directory array, or SID 0) is known as the root directory entry and is reserved for two purposes. First, it provides a root parent for all objects stationed at the root of the compound file. Second, its function is overloaded to store the size and starting sector for the mini stream.

The root directory entry behaves as both a stream and storage. All of the fields in the directory entry are valid for the root. The root directory entry's Name field typically contains the string "Root Entry" in Unicode, although some versions of structured storage (particularly the preliminary reference implementation and the Macintosh version) store only the first letter of this string, "R" in the name. This string is always ignored, since the root directory entry is known by its position at SID 0 rather than by its name, and its name is not otherwise used. New implementations should write "Root Entry" properly in the root directory entry for consistency and support manipulating files created with only the "R" name.

The class identifier (CLSID) stored in the root directory entry is normally used for Component Object Model (COM) activation of the document's application. The time stamps for the root storage are normally not maintained in the root directory entry. Rather, the root storage's creation and modification time stamps are normally stored on the file itself.

Other Directory Entries

Non-root directory entries are marked as either stream (STGTY_STREAM) or storage (STGTY_STORAGE) elements. Storage elements have a _clsid, _time[], and _sidChild values; stream elements may not. Stream elements have valid _sectStart, _ulSizeLow, _ulSizeHigh members, whereas these fields are set to zero for storage elements (except as noted above for the root directory entry).

To determine the physical file location of actual stream data from a stream directory entry, it is necessary to determine whether the stream exists in the FAT or the MiniFAT. Streams whose size is less than the _ulMiniSectorCutoff value (typically 4 KB) for the file exist in the mini stream. The _startSect is used as an index into the MiniFAT (which starts at _sectMiniFatStart) to track the chain of sectors through the mini stream. Streams whose size is greater than the _ulMiniSectorCutoff value for the file exist as standard streams—their _sectStart value is used as an index into the standard FAT which describes the chain of full sectors containing their data.

For 512-byte sectors, the _ulSizeHigh field should be set to zero, because this field was called the _pdtPropType in

older implementations. For 4 KB sectors, the _ulSizeHigh and _ulSizeLow are combined to form a 64-bit stream size.

Free (unused) directory entries are marked with STGTY_INVALID. The whole directory entry should consist of zeroes except for the child, right sibling, and left sibling pointers, which should be initialized to NOSTREAM.

Stream Sectors

Stream sectors are simply collections of arbitrary bytes. They are the building blocks of user streams, and no restrictions are imposed on their contents. Stream sectors are represented as chains in the FAT or MiniFAT, and each stream chain will have a single directory entry associated with it to hold its metadata (such as the name and size). The unused portion of the last sector of a stream should be filled with zeroes to avoid leaking unintended information.

Range Lock Sector

The range lock sector is the sector that covers file offsets 0x7FFFFF00-0x7FFFFFFF in the file, which are just before 2 GB. These offsets are reserved for byte-range locking to support concurrency, transactions, and other compound file features. The range lock sector is allocated in the FAT (marked with ENDOFCHAIN) when the compound file grows beyond 2 GB. Since 512-byte compound files are limited to 2 GB in size, these files do not need a range lock sector allocated. If the compound file is greater than 2 GB and then shrinks to below 2 GB, the range lock sector should be marked as FREESECT in the FAT.

This sector contains no data, and it should not be read or written to, since the byte-range locks that can be held might contain various offsets in the sector.

The reference implementation does not implement range locks for concurrent accesses of the same file, since it always opens with share-exclusive access. The only open-mode flags that do not require byte-range locks are read/write shared exclusive and read-only deny-write.

File Size Limits

Normal 512-byte sector compound files are limited to 2 GB in size for compatibility reasons. This means that a stream inside a 512-byte sector compound file is limited to less than 2 GB. Any attempt to grow the file or stream past 2 GB should result in an error such as STG_E_DOCFILETOOLARGE.

4 KB sector compound files can have 64-bit file and stream sizes, up to slightly less than 16 terabytes (4 KB x MAXREGSECT). SECT values above MAXREGSECT are reserved for denoting free sectors, end-of-chain, and other special sectors. The maximum number of directory entries is MAXREGSID (roughly 4 billion), since SID values above MAXREGSID are reserved for denoting free directory entries and other special values. This corresponds to a maximum directory stream of slightly less than 512 GB. The maximum size of the mini stream is MAXREGSECT x (1 << _uMiniSectorShift, typically 64 bytes), which is slightly less than 256 GB.

Any attempt to allocate a sector beyond MAXREGSECT or grow the directory stream beyond MAXREGSID entries should result in an error such as STG_E_DOCFILETOOLARGE.

Validation and Corruption

The compound file implementation should check the byte signature, the major version number in the header. If the major version number is greater than the currently supported version, or the byte signature does not match, or a reserved field is not zero, then an error such as STG_E_INVALIDHEADER should be returned.

The compound file implementation should reject all files with invalid or mismatched fields with an error such as STG_E_DOCFILECORRUPT. In addition, run time infinite loop detection is recommended to detect cyclic FAT, DIFAT, MiniFAT, mini stream, and directory chains. The actual sector count should match the size specified for a virtual stream. The actual directory entry name's string length should match the length specified in the directory entry.

It is important for the implementation to support 64-bit file/stream offsets and seek pointers to avoid truncation problems growing past 4 GB in size. It is important for the implementation to properly flush all the file changes to disk to avoid corruption issues. All new sectors and in-memory structures should be fully initialized to avoid security problems.

Examples

This section contains a hexadecimal dump of a sample structured storage file to clarify the binary file format.

The Header

```
_abSig              = DOCF 11E0 A1B1 1AE1
_clid               = 0000 0000 0000 0000 0000 0000 0000 0000
_uMinorVersion      = 003E
_uDllVersion        = 3
_uByteOrder         = FFFE (Intel byte order)
_uSectorShift       = 9 (512 bytes)
_uMiniSectorShift   = 6 (64 bytes)
_usReserved         = 0000     (must be zero)
_ulReserved1        = 00000000 (must be zero)
_csectDir           = 00000000 (not used for 512-byte sectors)
_csectFat           = 00000001 (1 Fat sector in this file)
_sectDirStart       = 00000001 (SECT #1 is first directory sector)
_signature          = 00000000 (not used, transactions not supported)
_ulMiniSectorCutoff = 00001000 (4096 bytes)
_sectMiniFatStart   = 00000002 (SECT#2 is first MiniFAT sector)
_csectMiniFat       = 00000001 (1 MiniFAT sector in this file)
_sectDifStart       = FFFFFFFE (no DIFAT, file is < 7Mb)
_csectDIF           = 00000000 (no DIFAT sectors in this file)
_sectFat[0]         = 00000000 (SECT#0 is the first FAT sector)
_sectFat[1..108]    = FFFFFFFF ... (continues with FFFFFFFF) (free FAT sectors)

    000000:  D0CF 11E0 A1B1 1AE1  0000 0000 0000 0000   ................
    000010:  0000 0000 0000 0000  3B00 0300 FEFF 0900   ........;.......
    000020:  0600 0000 0000 0000  0000 0000 0100 0000   ................
    000030:  0100 0000 0000 0000  0010 0000 0200 0000   ................
    000040:  0100 0000 FEFF FFFF  0000 0000 0000 0000   ................
    000050:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
    . . .
    0001F0:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
```

SECT #0: FAT Sector

This is the first and only FAT sector in the file.

```
SECT 0: FFFFFFFD = FATSECT: marks this sector as a FAT sector.
            Referred to in header by _sectFat[0]
SECT 1: FFFFFFFE = ENDOFCHAIN: marks the end of the directory chain,
            referred to in header by _sectDirStart
SECT 2: FFFFFFFE = ENDOFCHAIN: marks the end of the MiniFAT, referred to
            in header by _sectMiniFatStart
SECT 3: 00000004 = pointer to the next sector in the "Stream 1" data.
            This sector is the first sector of "Stream 1", it is referred
            to by the directory entry
SECT 4: ENDOFCHAIN (0xFFFFFFFE): marks the end of the "Stream 1" stream data.

Further Entries are empty (FREESECT = 0xFFFFFFFF)

    000200:  FDFF FFFF FEFF FFFF  FEFF FFFF 0400 0000   ................
    000210:  FEFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
    . . .
    0003F0:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
```

SECT #1: Directory Sector

This is the first and only directory sector in the file. This directory sector consists of four directory entries.

```
SID 0: Root SID: Root Name = "Root Entry"
SID 1: Element 1 SID: Name = "Storage 1"
SID 2: Element 2 SID: Name = "Stream 1"
SID 3: Unused
```

SID 0: Root Directory Entry

```
_ab          = (L"Root Entry")
_cb          = 0016 (22 bytes, includes the Unicode null terminator)
_mse         = 05 (STGTY_ROOT)
_bflags      = 01 (DE_BLACK)
_sidLeftSib  = FFFFFFFF (none)
_sidRightSib = FFFFFFFF (none)
_sidChild    = 00000001 (SID 1: "Storage 1")
_clsid       = 0067 6156 54C1 CE11 8553 00AA 00A1 F95B
_dwUserFlags = 00000000 (n/a for STGTY_ROOT)
_time[0]     = CreateTime   = 0000 0000 0000 0000 (none set)
_time[1]     = ModifyTime   = 801E 9213 4BB4 BA01 (??)
_sectStart   = 00000003 (starting sector of mini stream)
_ulSizeLow   = 00000240 (length of mini stream in bytes)
_ulSizeHigh  = 00000000

    000400:  5200 6F00 6F00 7400  2000 4500 6E00 7400   R.o.o.t. .E.n.t.
    000410:  7200 7900 0000 0000  0000 0000 0000 0000   r.y.............
    000420:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    000430:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    000440:  1600 0501 FFFF FFFF  FFFF FFFF 0100 0000   ................
    000450:  0067 6156 54C1 CE11  8553 00AA 00A1 F95B   .gaVT....S.....[
    000460:  0000 0000 0000 0000  0000 0000 801E 9213   ................
    000470:  4BB4 BA01 0300 0000  4002 0000 0000 0000   K.......@.......
```

SID 1: Storage 1

```
_ab          = (L"Storage 1")
_cb          = 0014 (20 bytes, including the Unicode null terminator)
_mse         = 01 (STGTY_STORAGE)
_bflags      = 01 (DE_BLACK)
_sidLeftSib  = FFFFFFFF (none)
_sidRightSib = FFFFFFFF (none)
_sidChild    = 00000002 (SID 2: "Stream 1")
_clsid       = 0000 0000 0000 0000 0000 0000 0000 0000 (none set)
_dwUserFlags = 00000000 (none set)
_time[0]     = CreateTime   = 00000000 00000000 (none set)
_time[1]     = ModifyTime   = 00000000 00000000 (none set)
_sectStart   = 00000000 (n/a)
_ulSizeLow   = 00000000 (n/a)
_ulSizeHigh  = 00000000 (n/a)

    000480:  5300 7400 6F00 7200  6100 6700 6500 2000   S.t.o.r.a.g.e. .
    000490:  3100 0000 0000 0000  0000 0000 0000 0000   1...............
    0004A0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    0004B0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    0004C0:  1400 0101 FFFF FFFF  FFFF FFFF 0200 0000   ................
    0004D0:  0061 6156 54C1 CE11  8553 00AA 00A1 F95B   .aaVT....S.....[
    0004E0:  0000 0000 0088 F912  4BB4 BA01 801E 9213   ........K.......
    0004F0:  4BB4 BA01 0000 0000  0000 0000 0000 0000   K...............
```

SID 2: Stream 1

```
_ab          = (L"Stream 1")
_cb          = 0012 (18 bytes, including the Unicode null terminator)
_mse         = 02 (STGTY_STREAM)
_bflags      = 01 (DE_BLACK)
_sidLeftSib  = FFFFFFFF (none)
```

```
_sidRightSib = FFFFFFFF (none)
_sidChild    = FFFFFFFF (n/a for STGTY_STREAM)
_clsid       = 0000 0000 0000 0000 0000 0000 0000 0000 (n/a)
_dwUserFlags = 00000000 (n/a)
_time[0]     = CreateTime   = 00000000 00000000 (n/a)
_time[1]     = ModifyTime   = 00000000 00000000 (n/a)
_startSect   = 00000000 (SECT in MiniFAT, since _ulSize is smaller
                         than _ulMiniSectorCutoff)
_ulSizeLow   = 00000220 (< ssheader._ulMiniSectorCutoff, so _sectStart
                         is in mini stream)
_ulSizeHigh  = 00000000 (n/a)

    000500:  5300 7400 7200 6500  6100 6D00 2000 3100   S.t.r.e.a.m. .1.
    000510:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    000520:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    000530:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    000540:  1200 0201 FFFF FFFF  FFFF FFFF FFFF FFFF   ................
    000550:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    000560:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    000570:  0000 0000 0000 0000  2002 0000 0000 0000   ........ .......
    000580:  0000 0000 0000 0000  0000 0000 0000 0000   ................
```

SID 3: Unused, Free

All fields are zeroes except for the child, right sibling, and left sibling pointers, which are set to NOSTREAM.

```
    000590:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    0005A0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    0005B0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    0005C0:  0000 0000 FFFF FFFF  FFFF FFFF FFFF FFFF   ................
    0005D0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    0005E0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
    0005F0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
```

SECT #2: MiniFAT Sector

The MiniFAT sector is identical to a FAT sector in structure, but instead of describing allocations for the file, the MiniFAT describes allocations for the mini stream. Here is a chain of 8 contiguous sectors.

```
SECT 0: 00000001: pointer to the second sector in the "Stream 1" data.
            This sector is the first sector of "Stream 1", it is referred
            to by _sectStart of SID 2
SECT 1: 00000002: pointer to the third sector in the "Stream 1" data.
            This sector is the second sector of "Stream 1", it is referred
            to in MiniFat SECT 0, above.
. . .
SECT 8: FFFFFFFE = ENDOFCHAIN: marks the end of the "Stream 1" data.

Further Entries are empty (FREESECT = 0xFFFFFFFF)

    000600:  0100 0000 0200 0000  0300 0000 0400 0000   ................
    000610:  0500 0000 0600 0000  0700 0000 0800 0000   ................
    000620:  FEFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
    . . .
    0007F0:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
```

SECT #3: Mini Stream Sector

The mini stream contains data for all streams whose length is less than the header's _ulMiniSectorCutoff, which is normally 4 KB. In this example, the mini stream contains the data for Stream 1. The unused portion of the sector is zeroed out.

```
    // referred to by SECTs in MiniFat of SECT 3, above

    000800:  4461 7461 2066 6F72  2073 7472 6561 6D20   Data for stream
    000810:  3144 6174 6120 666F  7220 7374 7265 616D   1Data for stream
    000820:  2031 4461 7461 2066  6F72 2073 7472 6561    1Data for strea
```

```
. . .
000A00:  7461 2066 6F72 2073  7472 6561 6D20 3144   ta for stream 1D
000A10:  6174 6120 666F 7220  7374 7265 616D 2031   ata for stream 1

// although data ends at 000A1F, MiniSector is filled to the end with
// known data (such as zeros) to prevent random disk or memory contents
// from contaminating the file on-disk.

000A20:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000A30:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000A40:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000A50:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000A60:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000A70:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000A80:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000A90:  0000 0000 0000 0000  0000 0000 0000 0000   ................
. . .
000BF0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
```