Microsoft ™ Tape Format Specification

Version 1.00a - document rev. 1.8

September 15, 2000

derived from Microsoft ™ Tape Format Version 1.0

Document Revision 1.0

Revision History

Date	Description
03-13-98	Changed copyright notice.
10-21-97	Changed File Attributes. The file attributes were inadvertently changed to incorrect values in a previous version of the specification. The changes to the file attributes reflect the correct definition.
09-03-97	The Media Catalog Version field of the Start of Set Descriptor Block now refers to the Media Based Catalog definition for assigned values.
07-10-97	Added clarifications to the Common Block Header fields of the Tape, End of Set, End of Media, and Soft Filemark Descriptor Blocks.
06-10-97	Added sparse file support and NT specific streams. Add NT OS Specific Information for volumes.
04-11-97	Removed STREAM_CONTINUE reference from Variable Length Stream.
04-02-97	Corrected text in TAPE Attributes field of the Tape Header Descriptor Block.
04-02-97	Corrected NT OS Specific Information (OS ID Number 14, OS Version Number 0). Reserved field was added to correct alignment problem.
04-02-97	Added NT OS Specific Information (OS ID Number 14, OS Version Number 1).

Print History

Date	Version
03-12-98	Version 1.00a Rev 1.8
10-21-97	Version 1.00a Rev 1.7
07-10-97	Version 1.00a Rev 1.6
06-10-97	Version 1.00a Rev 1.5
04-10-97	Version 1.00a Rev 1.4
04-02-97	Version 1.00a Rev 1.3
08-20-96	Version 1.00a Rev 1.2

Table of Contents

Table of Figures

1. Introduction

This document describes the logical data format used in Microsoft Tape Format (MTF). Media types which can use this data format include magnetic tapes of many types (QIC, 4mm DAT, 8mm, DLT, etc.), optical disks (Power Drive, CD-ROM), magnetic disks, etc. MTF is used while writing and reading data to and from removable storage devices during storage management or data protection operations such as data transfers, copies, backup and restore.

Throughout this document, the term "tape" is used when referring to the removable media. Tapes are shown in most of the diagrams depicting the physical layout of data. Even some of the data structures include the name "TAPE". Keep in mind that disk based media is equally suitable for MTF and tape is used only as an example, and because this specification originated as a tape format specification before optical disk media became a viable solution for storage management.

This format is compatible with the data format used in the ***NT Backup*** applet program that comes bundled with Microsoft® Windows NT™ version 3.X and 4.X.

1.1 Who Should Read This Document

This document should be read by anyone who needs to understand or implement the Microsoft Tape Format. It is expected that the reader have a general knowledge of storage management operations, tape drives and file systems. Knowledge of tape data formats is helpful but not required.

1.2 Document Layout

This document is Revision 1.00a of the Microsoft Tape Format Specification. It is a refinement of the Microsoft Tape Format Version 1.0 Specification. While the design of MTF Version 1.00a will remain unchanged, future updates and revisions to this document will continue to be made in an effort to describe the specification as clearly and accurately as possible.

Section 1 ***Introduction***, provides an introduction to the Microsoft Tape Format (MTF).

Section 2 ***Design Goals***, describing the capabilities inherent in the MTF.

Section 3 ***Format Description***, provides a broad look at the organization of MTF, covering material such as Data Sets, the fundamental building blocks of MTF called "Descriptor Blocks", the use of data streams, the Media Based Catalog, filemarks, the physical and logical characteristics of the format, and spanning Data Sets across multiple media (tapes or disks).

Section 4 ***Support Structures***, provides detailed definition of support structures used throughout MTF.

Section 5 ***Descriptor Blocks***, provides detailed definition of Descriptor Blocks.

Section 6 ***Data Streams***, provides detailed definitions of Data Streams.

Section 7 ***Media Based Catalog***, provides detailed definitions of Type 1 and Type 2 Media Based Catalogs.

Section 8 ***End Of Media Processing***, provides a detailed description of End Of Media Processing.

The ***Appendices***, include detailed information on Operating System Specific Data, Password Encryption Algorithm, Data Compression Algorithm, and Implementation Issues.

2. Design Goals

This sections describes the design goals employed in the development of the Microsoft Tape Format.

- Fast retrieval of stored data.
- Low processing overhead to ensure optimum performance on low-end systems and devices. This is accomplished by careful design of the control structures to reduce the amount of interpretation the application software needs to do.
- Allows applications to ignore information on the media that is not understood by the target operating system. This feature makes it possible to restore data across platforms (e.g., data backed up on an Apple Macintosh system may be restored to a DOS system, ignoring the resource fork which DOS does not understand).
- The ability to extend the format for specialized processing by adding new DBLKs and data streams without rendering the format unreadable by other applications. Applications which are not aware of the extensions can easily skip over them, both increasing backward/forward compatibility, and allowing the restoration of data from media created by another vendor's application.
- Data structures are arranged so that 32-bit values are aligned on 32-bit boundaries, and 16-bit values are aligned on 16-bit boundaries. This is important because some processors require this alignment to run at maximum efficiency. By making sure this alignment is followed it is easier for the implementor to map these structures directly onto data buffers.
- Reliable end of media handling.
- The ability to restore **any** remaining portion of a Data Set which spans multiple media (tapes or disks) in the event one or more media is lost or damaged.
- Format support to deal with corrupt files encountered on the primary storage volume that is being written to removable media.
- Support for unlimited directory path and file name lengths.
- 64-bit file data sizes.
- Allows the application to take full advantage of a drive's capabilities (e.g., Block Seek, Fast Seek to End of Data, etc.) without hindering less capable drives.

3. Format Description

This section presents an overview of the Microsoft Tape Format (MTF). It discusses the fundamental elements that are used in data management operations and how they are organized. This information provides the foundation from which the rest the this document will build.

3.1 Data Sets

When a collection of objects are written to removable storage media (tape, optical disk, etc.) during a data management operation such as a backup, transfer or copy, it is stored as a ***Data Set***. A medium may contain more than one Data Set and a Data Set may span from one medium to another. The diagrams that follow use tapes as the medium type. The term ***Media Family*** refers to a collection of one or more Data Sets appended together and spanning one or more individual tapes or *media*. The diagram below is a simplified picture of how Data Sets are placed on one or more media.

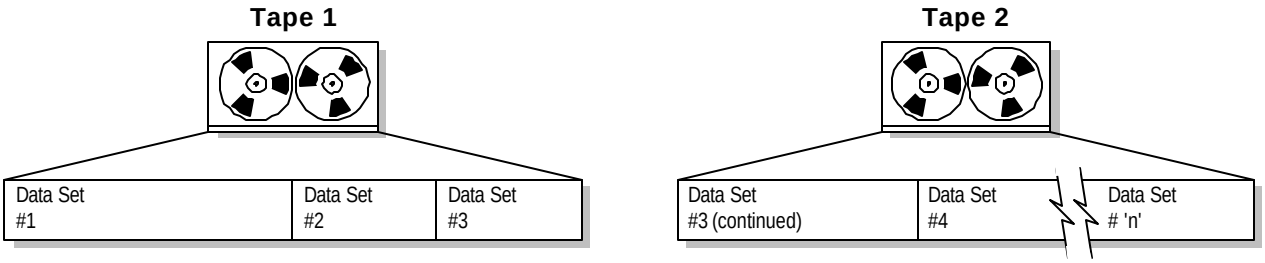


Figure 1. Media Family

3.2 Fundamental Elements

The fundamental elements of MTF are ***Descriptor Blocks***, ***Data Streams***, and ***Filemarks***. Descriptor Blocks are used for format control, Data Streams are associated with Descriptor Blocks to provide data encapsulation and alignment, and filemarks are used for logical separation and fast positioning within a media.

3.2.1 Descriptor Blocks

Descriptor Blocks are the primary building blocks on which MTF is founded. Throughout this document a Descriptor Block will be abbreviated to DBLK. MTF defines many DBLKs each of which is uniquely suited for the role it was defined.

3.2.1.1 Descriptor Block Anatomy

A DBLK is essentially a variable length block of data that is divided into four parts. The first is the Common Block Header which is fixed length structure that is common to all DBLKs. The second is the Fixed Length DBLK Information that is specific to the type of DBLK being defined. The third is the Operating System Specific Data that is defined based on the type of DBLK and Operating System. The fourth and last is the Variable Length DBLK Specific Information which contains variable length that cannot be stored with the Fixed Length DBLK Information. Of the four parts listed, only the Common Block Header is required.

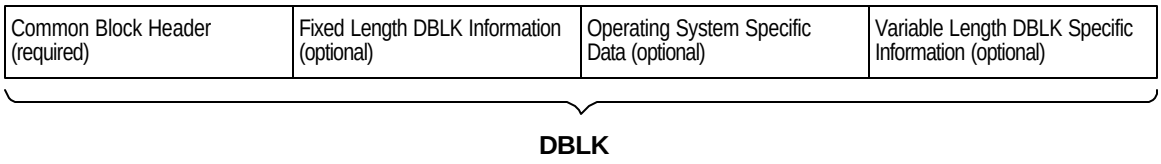


Figure 2. Descriptor Block Layout

3.2.1.1.1 Common Block Header

The Common Block Header is at the beginning of each DBLK and includes general information about the DBLK. The Common Block Header includes a link to the Operating System Specific Data section. For a detailed description of the Common Block Header, see the Chapter on ***Descriptor Blocks.***

3.2.1.1.2 Fixed Lenght DBLK Information

The Fixed Length DBLK Information follows the Common Block Header and contains information that is unique to the type of DBLK defined. The Fixed Length DBLK Information is a fixed length for each DBLK type and may contain links into the Variable Length DBLK Specific Information section. The Fixed Length DBLK Information is optional and may be omitted if the defined DBLK type contains no unique information. For a detailed description of the defined Descriptor Blocks, see the Chapter on ***Descriptor Blocks.***

3.2.1.1.3 Operating System Specific Data

The Operating System Specific Data may optionally contain variable length information that is specific to the type of DBLK and Operating System. For a detailed description of Operating System Specific Data, see ***Appendix A***.

3.2.1.1.4 Variable Length DBLK Specific Information

The Variable Length DBLK Specific Information may optionally contain variable length information that is specific to the type of DBLK. The information stored in the Variable Length DBLK Specific Information is referenced through links in the Fixed Length DBLK Information.

3.2.1.1.5 Detailed Descriptor Block Layout

The following figure shows a detailed layout of a generic Descriptor Block. The MTF Tape Address sub-structure in the Common Block Header is the link to the Operating System Specific Information section. The MTF Tape Address sub-structure in the Fixed Length DBLK Specific Information is the link(s) to the Variable Length DBLK Specific Information section.

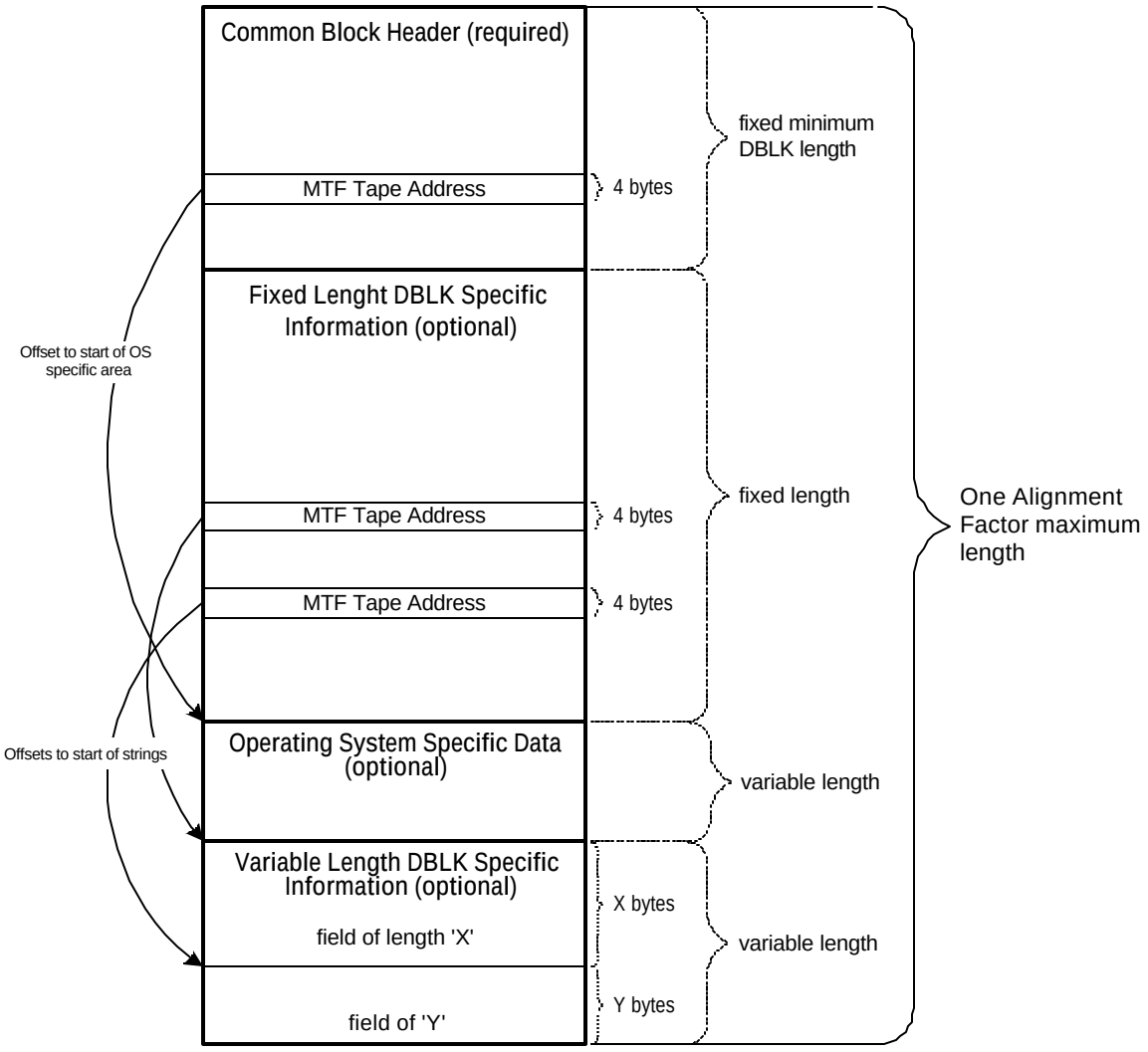


Figure 3. Detailed Descriptor Block

3.2.1.2 Defined Descriptor Blocks

MTF defines a number of Descriptor Blocks which are used to control the placement of data in a data management operation. The following is a general overview the DBLKs currently defined for MTF. New DBLKs will be defined by the MTF Review Committee in the future as the need arises.

3.2.1.2.1 Tape Header Descriptor Block

The *Tape Header Descriptor Block* (MTF_TAPE DBLK) is located at the front of each media. The MTF_TAPE DBLK describes the contents of the media. This information includes a unique identifier to indicate the Media Family to which the media belongs, the sequence number of the media in the Media Family, and a name string for user identification, as well as other information needed to interpret the data on the media. Information about the presence, and type of Media Based Catalogs is available here.

Note: Despite the name of this DBLK, it is used on tape, optical disk, or other types of removable storage media.

3.2.1.2.2 Start of Data Set Descriptor Block

The *Start of Data Set Descriptor Block* (MTF_SSET DBLK) is located at the front of each Data Set. It contains information that describes the Data Set such as the name, a user description, the password, a sequence number, the date and time that data began being written to media, and the type of data management operation (transfer, copy, normal backup, differential backup, etc.) used to create the Data Set.

3.2.1.2.3 Volume Descriptor Block

The *Volume Descriptor Block* (MTF_VOLB DBLK) describes a volume which is being written to the media. This includes the device name, volume name, machine name and media write date.

3.2.1.2.4 Directory Descriptor Block

The *Directory Descriptor Block* (MTF_DIRB DBLK) describes the full path of the directory being written to media. This includes the directory name, the directory creation date and time, the last modification date, backup date, last access date and directory attributes such as read only.

3.2.1.2.5 File Descriptor Block

The *File Descriptor Block* (MTF_FILE DBLK) describes the file which is being written to media and is followed by the actual file data. The MTF_FILE DBLK contains information such as the file name, file size, the date and time the file was created, last accessed and last modified, and file attributes such as read only, hidden, system, etc.

3.2.1.2.6 Corrupt Object Descriptor Block

It is often the case that a DBLK has already been written when it is discovered that not all of its associated data can be read due to disk corruption, network failure, etc. When this condition occurs, the portions of the stream that could not be read are padded to maintain the correct stream size.

A *Corrupt Object Descriptor Block* (MTF_CFIL DBLK) is then written to indicate that the data associated with the previous DBLK is corrupt. The MTF_CFIL DBLK contains fields for the stream number and the byte offset in that stream where the corruption began.

3.2.1.2.7 End of Set Pad Descriptor Block

The *End of Set Pad Descriptor Block* (MTF_ESPB DBLK) is only used when the physical block size of the device is larger than the format logical block size. The MTF_ESPB DBLK is an optional method used at the end of a Data Set to fill the gap to the next physical block boundary. Alternately, the SPAD associated with the last DBLK can be extended to the next physical block boundary.

Note: Format Logical Block is defined in section 3.7.

3.2.1.2.8 End of Set Descriptor Block

All Data Sets end with the *End of Set Descriptor Block* (MTF_ESET DBLK). Since it is not necessarily known how many objects will be written in a Data Set when the operation begins, MTF_ESET serves as an indicator to show that the preceding filemark indicates the end of the Data Set. It also contains information which isn't available until the data management operation is complete, such as the number of corrupt objects written to the media.

3.2.1.2.9 End of Tape Marker Descriptor Block

The *End of Tape Marker Descriptor Block* (MTF_EOTM DBLK) is the last DBLK written to a "full" media. As with the MTF_ESET, the MTF_EOTM serves primarily as an indicator, but does contain information necessary for fast access of Media Based Catalogs.

3.2.1.2.10 Soft Filemark Descriptor Block

The *Soft Filemark Descriptor Block* (MTF_SFMB DBLK) is used to emulate filemarks when hardware support is not available.

3.2.2 Data Streams

Data Streams are used to encapsulate data. This encapsulated data can then be associated with a Descriptor Block. A Data Stream is comprised of a Stream Header followed by the Steam Data. A field within the Stream Header defines the type of Stream Data that will follow. Only one type of Stream Data can be encapsulated by a Stream Header. The segregation of different Stream Data types provides a means of separating platform independent data from platform specific data. For a detailed description, see the section on ***Data Streams***.

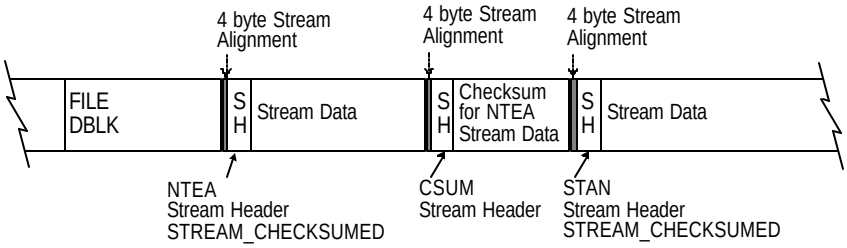


Figure 4. Data Streams

3.2.3 Filemarks

Filemarks are used for logical separation and fast positioning within a media. If the device being used does not provide filemarks, the filemarks must be emulated by a device driver or by use of the Soft Filemark Descriptor Block. The placement of filemarks is discussed in the next section Media Layout.

3.3 Media Layout

MTF defines that a media be divided into a Media Header, one or more Data Sets, and End of Media. The Media Header is used to uniquely identify the media. The Data Set is used to store a collection of Descriptor Blocks and Data Streams used in a data management operation. The End of Media is used to span from one media to the next.

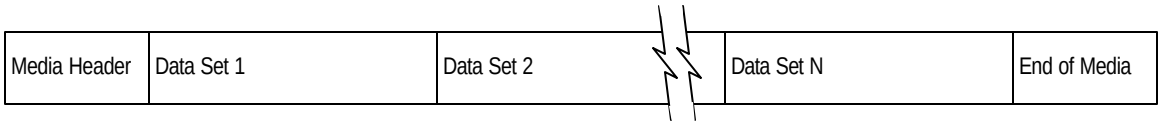


Figure 5. Media Layout

3.3.1 Media Header

The Media Header is used to uniquely identify the media. A Media Header is comprised of a Tape Header Descriptor Block (MTF_TAPE DBLK), SPAD Data Stream, and filemark. The SPAD Data Stream is used to fill the gap between the MTF_TAPE DBLK and filemark.

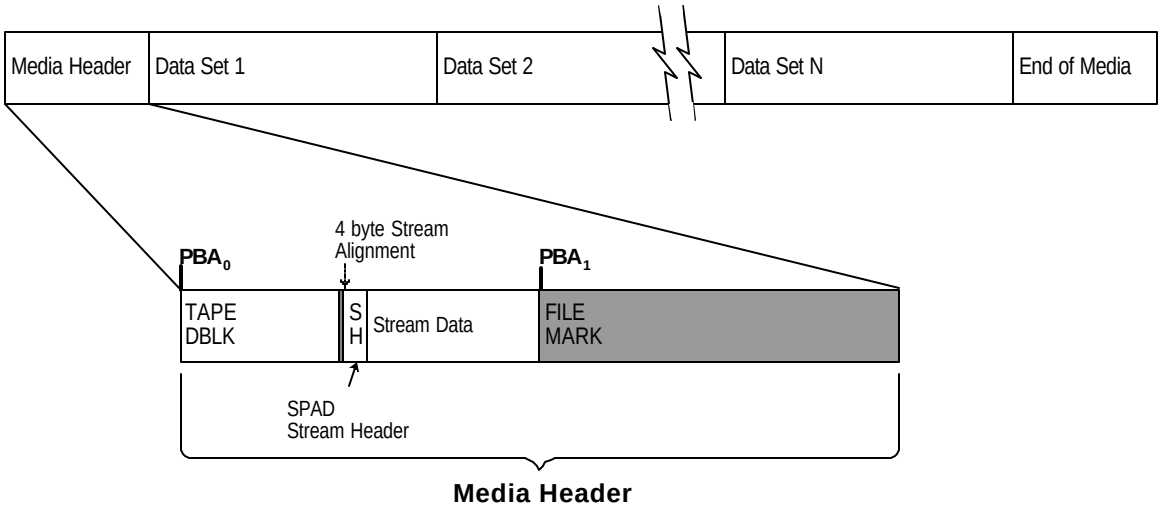


Figure 6. Media Header

3.3.2 Data Sets

A Data Set is comprised of a Start Of Data Set Descriptor Block (MTF_SSET DBLK), the Descriptor Blocks used for the data management operation, a filemark, End Of Set Descriptor Block (MTF_ESET DBLK), and filemark. Typical Descriptor Blocks used for a data management operation include the Volume, Directory, and File Descriptor Blocks. The End Of Set Descriptor may optionally have Media Based Catalog Data Streams associated with it.

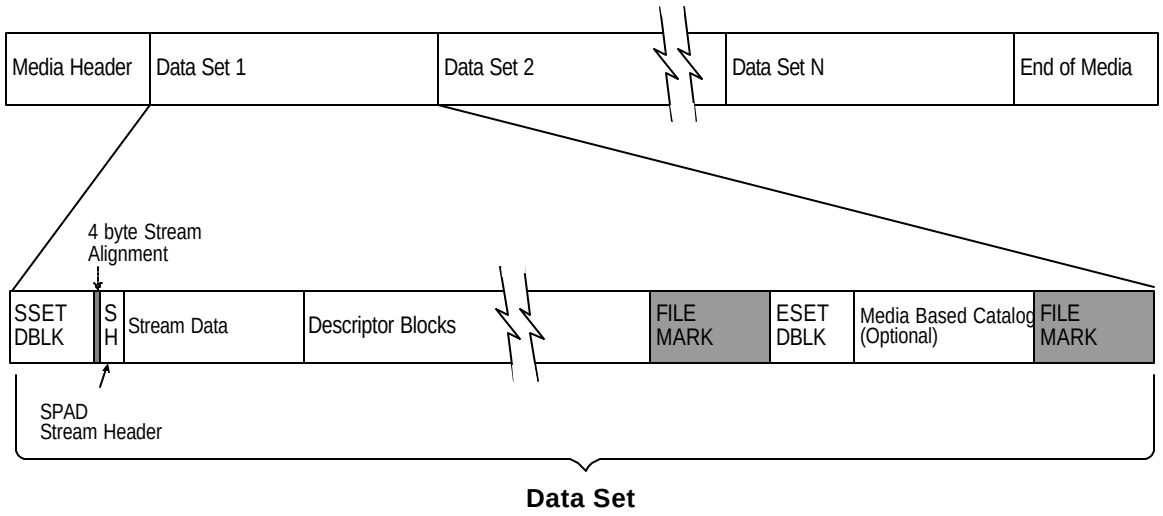


Figure 7. Data Set

3.3.2.1 Implied Precedence within a Data Set

It is important to understand that MTF is a linear format and uses ***Implied Precedence*** to preserve the parent child relationship between Descriptor Blocks. That is, the parent child relationship is implied in the definition of the Descriptor Block and cannot be determined when an unknown Descriptor Block is encountered.

Table 1. Implied Precedence within a Data Set

	Parent	Child
MTF_SSET		MTF_VOLB
MTF_VOLB	MTF_SSET	MTF_DIRB
MTF_DIRB	MTF_VOLB	MTF_FILE
MTF_FILE		

Take for example the Volume Descriptor Block. Once the Volume Descriptor Block is written to a Data Set, all Directory and File Descriptor Blocks that follow are children of that Volume Descriptor Block until another Volume Descriptor Block is written. The same is also true of the Directory Descriptor Block. When a Directory Descriptor Block is written to a Data Set, all File Descriptor Blocks that follow are children of that Directory Descriptor Block until another Directory Descriptor Block is written.

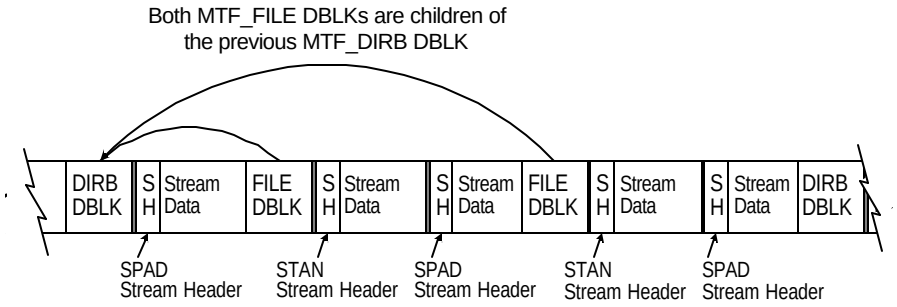


Figure 8. Implied Precedence

3.3.2.2 Media Based Catalogs

The Media Based Catalog provides a quick method of locating Data Sets and specific objects within each Data Set. The abbreviation "MBC" is used throughout this specification to refer to Media Based Catalog. The MBC consists of two parts, a ***File/Directory Detail*** that provides specific information about the contents of a single Data Set, and the ***Set Map*** which provides cumulative information about all the Data Sets on a Media Family. Both the File/Directory Detail and the Set Map are stored as data streams associated with the MTF_ESET. For a detailed description see the section on ***Media Based Catalogs***.

A Set Map may exist on tape without a File/Directory Detail. However, a File/Directory Detail can only exist if a Set Map is also present. The File/Directory Detail and Set Map must be data streams associated with a single MTF_ESET.

The Microsoft Tape Format has been designed not to require the use of Media Based Catalogs for access to objects stored on removable media. However, the use of the MBC is strongly recommended because it provides much faster access to information about objects on the media and to the actual data objects themselves.

3.3.3 End of Media

When the End of Media (tape or disk) is reached while writing a Data Set, a filemark is placed on the medium followed by an MTF_EOTM DBLK and another filemark. The write operation continues on the next medium which is called a "continuation" medium. The process of continuing the Data Set from one medium to another is called "spanning".

The figure below is an example of a Data Set containing directories and files that spans from one tape to another in the middle of a data stream following a MTF_FILE DBLK. The span point occurs at Format Logical Address 154 in the data stream for FILE "R". You can see that the data stream continues at FLA 154 on the next tape. Only the unwritten portion of the data stream is placed on the next medium. In this example, the physical block size is 1024 bytes, filemarks are the same length as the physical block size, and the Format Logical Block size is 512 bytes.

Continuation DBLKs are necessary on the continuation medium. The MTF_SSET, MTF_VOLB, MTF_DIRB, and MTF_FILE DBLKs that describe the spanning set, volume and directory and file respectively must be repeated on the continuation tape. Notice that the Physical Block Addresses do not continue on the second tape like the Format Logical Addresses do. This is because PBAs are controlled by the tape device which knows nothing of spanning, whereas Format Logical Addresses are controlled by MTF and are continuous for an entire Data Set regardless of the number of tapes required to hold it. The Format Logical Address of the continuation DBLKs must be calculated using the Format Logical Address at the span point and the number of continuation DBLKs that precede the continuation span point. For example, the FLA of the MTF_VOLB DBLK (151) on the continuation tape is calculated by subtracting 3 from the FLA of the continuation span point (154).

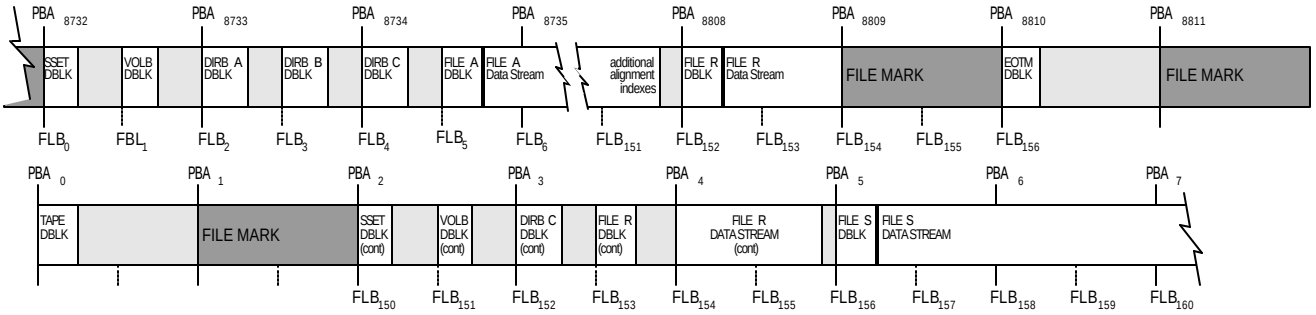


Figure 9. The Effect of Spanning Physical Block Address and Format Logical Addresses

The above example is just one way in which an End Of Media condition can occur. Appendix J is devoted to the different End Of Media conditions and how spanning is handled in MTF. There are actually two methods of handling the continuation DBLKs. The first method, which is shown in the above example, uses a single MTF_DIRB DBLK (DIRB C) prior to the span point. This is allowed because the path information with the nested directory structure is contained in every MTF_DIRB DBLK. The second method, not shown here, includes every MTF_DIRB DBLK (DIRB A, DIRB B, DIRB C) describing each element of the path at the span point.

3.4 Addressing

This section describes what a physical block is and how physical block addressing is performed as well as format logical block addressing and how to calculate a physical block address of a given Descriptor Block in a Data Set.

3.4.1 Physical Block

The term "physical block" refers to the minimum number of bytes which can be written to the removable storage medium by the device. The size of a physical block varies from one device to the next. Many tape devices now offer the ability to request the current position of the tape in terms of a physical block offset. We refer to this position in MTF as *a **Physical Block Address***, abbreviated **PBA**. These devices also have the ability to seek to a given PBA at a much faster rate than older tape devices which required rewinding and reading out to the same position.

MTF requires the ability to calculate the PBA of a given object in a Data Set given the PBA of the start of the Data Set, and the data offset of the object. In other words, PBAs between filemarks must be sequential. It is also required that all device drivers for a given device report the same PBA for any given location on the media. Given a PBA, all device drivers must seek to the same location on the media. Note that many devices do not directly support calculating PBAs in this manner. As a result, it is up to the software and/or device driver to ensure this. Refer to Appendix L for details on how this is accomplished on a specific device.

When a device writes a physical block to media, the physical block typically contains header information, data from the host computer, CRC information and ECC. The header, CRC and ECC information are automatically added by the device. The Microsoft Tape Format is only concerned with the data portion of the physical block that is written by the device and its physical address (PBA). When we talk about a Physical Block Address, we are referring to the address used by the tape device to identify and locate the data contained within that physical block.

3.4.2 Format Logical Block

If the PBA of every DBLK written to tape were stored in a catalog on primary disk, it would greatly reduce the time required to restore selected files from various places on the storage media. However, this method poses a problem: it would require all DBLKs to be aligned on physical block boundaries, resulting in a significant waste of space. On a device with a physical block size of eight kilobytes, several kilobytes of wasted space would result for each DBLK written. It would also require requesting this position prior to writing each object which would increase the time required to perform the write operation. For this reason, the concept of a ***Format Logical Block*** is introduced. Figure 3-6 shows several DBLKs belonging to the Data Set located within just two physical blocks with little wasted space. This is possible through the use of a Format Logical Block.

The MTF_SSET DBLK at the beginning of every Data Set always follows a filemark and thus is always aligned on a physical block boundary. Its PBA is obtained from the tape device driver and stored within the MTF_SSET DBLK. All DBLKs must be aligned on a Format Logical Block boundary. The size of the Format Logical Block can be 512 or 1024 bytes in MTF Ver. 1.00a. The Format Logical Block size that will be used for a specific medium is written in the MTF_TAPE DBLK at the beginning of the medium and must be consistent for the entire length of the medium and across an entire Data Set if it spans media. For example, if 512 bytes is chosen for a tape and the Data Set spans to another tape, the Format Logical Block size on the next tape must also be 512 bytes, not 1024 bytes.

The location of a specific DBLK can be found within a Data Set by using an address for the Format Logical Block that the DBLK starts on. This address is called the Format Logical Address and is abbreviated **FLA**. The Format Logical Address is the number of Format Logical Blocks from the start of the Data Set and can be thought of as a zero-based index into a Data Set. The Format Logical Address is a 64-bit unsigned integer. Every DBLK found in a Data Set has a unique FLA that is stored in the Common Block Header structure of the DBLK itself and in the File/Directory Detail portion of the Media Based Catalog. When restoring an object (volume, directory, or file), the FLA can be used in conjunction with the PBA of the MTF_SSET to calculate and seek to the exact location of the desired object's DBLK. For example, the MTF_VOLB DBLK in Figure 3-6 can be precisely located using the PBA of “y” and the FLA of “1”.

The flexibility within MTF to determine the Format Logical Block size allows vendors to choose a size that optimizes the speed or storage capacity for the particular storage device being used, and the type of data being written. The smaller Format Logical Block of 512 bytes results in less wasted space on the media as opposed to the 1024 bytes Format Logical Block.

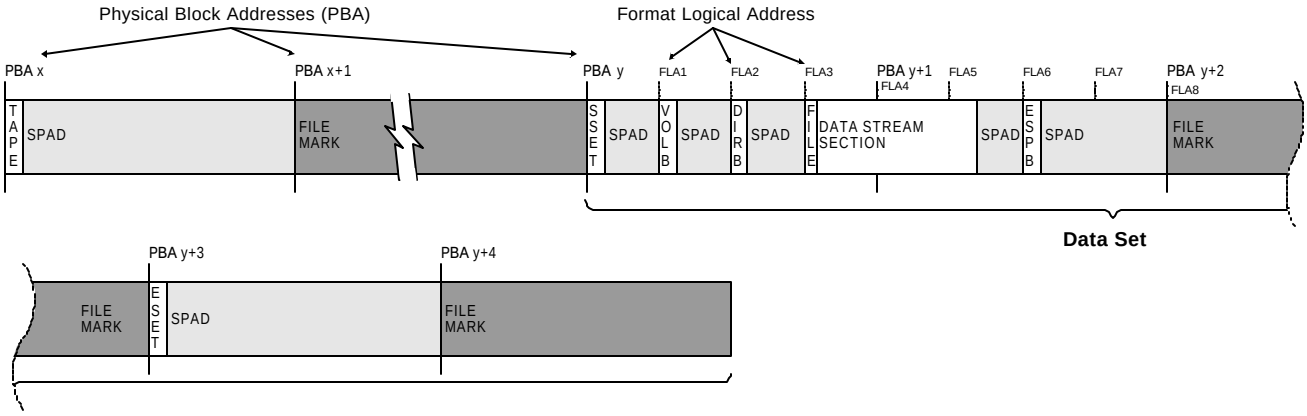


Figure 10. Physical Block and Format Logical Block Boundaries

3.4.3 Calculating Physical Block Addresses

One of the MTF design goals was fast retrieval of stored data. Every Descriptor Block in a Data Set contains a Format Logical Block Address. The following calculation provides the means of determining the Physical Block Address of a Descriptor Block given the Format Logical Block Address it contains. The result of the calculation is rounded down.

$$Req_{PBA} = (Req_{FLA} - SSET_{FLA}) / (\text{Physical Block Size} / \text{Format Logical Block Size}) + SSET_{PBA}$$

Req_{PBA}	is the Physical Block Address of the requested Descriptor Block.
Req_{FLA}	is the Format Logical Block Address of the requested Descriptor Block.
$SSET_{FLA}$	is the Format Logical Block Address of the MTF_SSET Descriptor Block (non zero on spanned next media of a spanned data set).
$SSET_{PBA}$	is the Physical Block Address of the MTF_SSET Descriptor Block.

Figure 11. Calculating Physical Block Addresses

3.5 Alignment

MTF is a linear format that must have Descriptor Blocks and Data Streams aligned on specific boundaries. Descriptor Blocks are aligned to Format Logical Block Boundaries and Data Stream are aligned to a Stream Alignment Factor. Filemarks are aligned to a Physical Block Boundary.

3.5.1 Descriptor Blocks

Descriptor Blocks must be aligned on a Format Logical Block Boundary. To do this, all Descriptor Blocks use the SPAD Data Stream as the last Data Stream associated with the Descriptor Block.

3.5.2 Data Streams

All Data Streams are aligned on a Stream Alignment Factor of four bytes. A fill pattern of zero is used in the four byte Stream Alignment for C2 security. If the Data Stream is already on a Stream Alignment Factor, no pad is needed. In the figure below, the File Descriptor Block contains a field Offset To Next Event. This field contains the size of the File Descriptor Block plus the number of bytes necessary to align the Data Stream to a four byte Stream Alignment Factor.

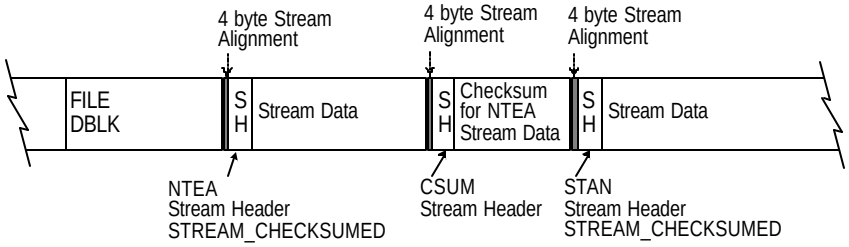


Figure 12. Stream Alignment Factor

3.5.3 Filemarks

Filemarks are always written on a Physical Block Boundary. The SPAD Data Stream is always used to pad to the next Physical Block Boundary where the filemark can be written. The End of Set Pad Descriptor Block may optionally be used in a Data Set prior to the first filemark.

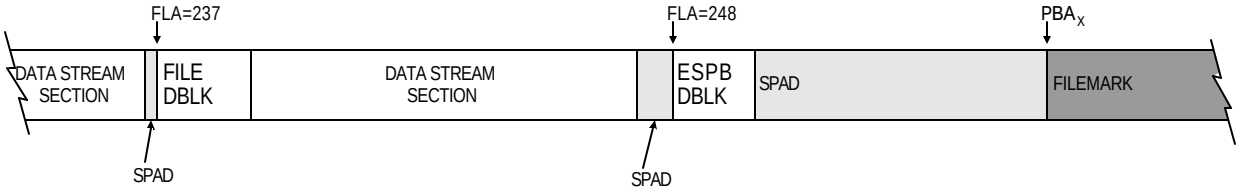


Figure 13. With End of Set Pad Descriptor Block

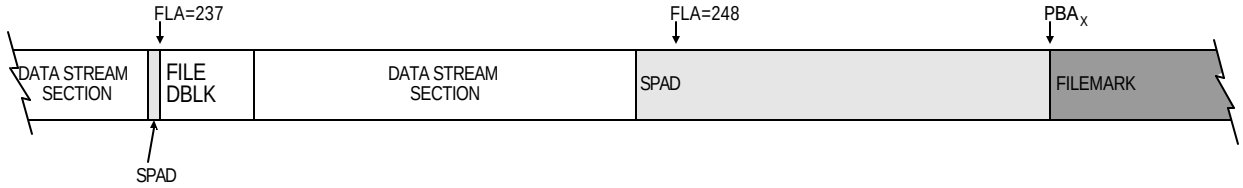


Figure 14. Without End of Set Pad Descriptor Block

4. Support Structures

This section provides detailed information about the support structures used in the higher level MTF structures. These support structure are an integral part of DBLKs and *Stream Header*s. The three support structures described below are comprised of fields of specific length, each having specific functions. They are building blocks used by the higher level structures.

Note: *All multi-byte entities are written in **INTEL** (little endian) format*

4.1 UINT64

This low level structure provides a method for specifying an unsigned 64-bit integer value within a DBLK structure.

Offset	Content	Type	Size
0 *0h*	Least Significant 32-bits	UINT32	4 bytes
4 *4h*	Most Significant 32-bits	UINT32	4 bytes

Structure 1. UINT64

4.2 MTF_TAPE_ADDRESS

The MTF_TAPE_ADDRESS low level structure is used inside the Common Block Header structure and inside many of the DBLK structures to identify non-fixed length information. The MTF_TAPE_ADDRESS low level structure is 4 bytes in length consisting of two 2 byte fields. The first field (*Size*) defines the size of the variable length field being referenced. The second field (*Offset*) contains an offset to the start of the field from the beginning of the structure containing the MTF_TAPE_ADDRESS.

Offset	Field Name	Type	Size
0 *0h*	Size	UINT16	2 bytes
2 *2h*	Offset	UINT16	2 bytes

Structure 2. MTF_TAPE_ADDRESS

4.3 MTF_DATE_TIME

The MTF_DATE_TIME low level structure uses a single 5 byte field containing a date and a time with resolution down to the second. One way this structure is used is in the MTF_FILE and MTF_DIRB DBLKs to define specific points in time when files and directories were created, modified, etc. The MTF_DATE_TIME low level structure is defined as follows. An unknown or undefined date and time is represented by using zero for all five bytes.

Offset	Content	Type	Size
0 *0h*	40-bit packed date and time as shown below.	UINT8[5]	5 bytes

Structure 3. MTF_DATE_TIME

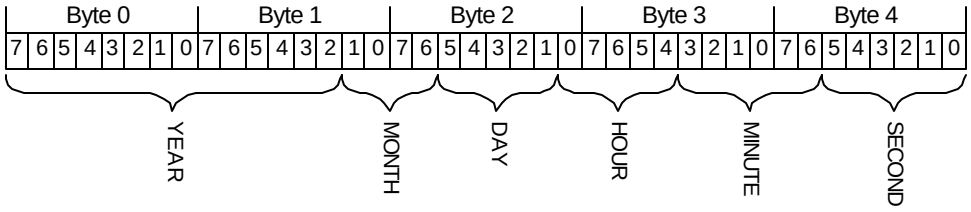
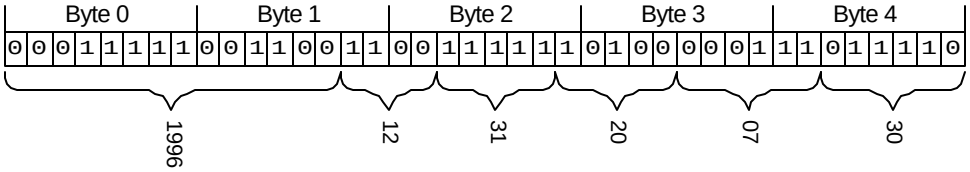


Figure 15. Bitwise Organization of MTF_DATE_TIME



Date = 12/31/1996
Time = 20:07:30

Figure 16. Example Data and Time in MTF_DATE_TIME Format

5. Descriptor Blocks

This section provides detailed information about Descriptor Blocks (DBLKs). A more general description of the ways in which the DBLKs are used in MTF can be found in Section 3, ***Format Description,*** without the detail covered here.

5.1 Common Block Header

The *Common Block Header* (MTF_DB_HDR) structure is found at the start of each DBLK. The MTF_DB_HDR contains general information required for each DBLK and includes fields describing the type of DBLK, its attributes (continuation, compression, presence of MBC, etc.), operating system specific information and the displayable size of the object defined by the DBLK (e.g. file size).

Offset	Field Name	Type	Size
0 *00h*	DBLK Type	UINT32	4 bytes
4 *04h*	Block Attributes	UINT32	4 bytes
8 *08h*	Offset To First Event	UINT16	2 bytes
10 *0Ah*	OS ID	UINT8	1 byte
11 *0Bh*	OS Version	UINT8	1 byte
12 *0Ch*	Displayable Size	UINT64	8 bytes
20 *14h*	Format Logical Address	UINT64	8 bytes
28 *1Ch*	Reserved for MBC	UINT16	2 bytes
30 *1Eh*	Reserved	- - -	6 bytes
36 *24h*	Control Block ID	UINT32	4 bytes
40 *28h*	Reserved	- - -	4 bytes
44 *2Ch*	OS Specific Data	MTF_TAPE_ADDRESS	4 bytes
48 *30h*	String Type	UINT8	1 byte
49 *31h*	Reserved	- - -	1 byte
50 *32h*	Header Checksum	UINT16	2 bytes

Structure 4. Common Block Header (MTF_DB_HDR)

DBLK Type {4 bytes}

The *DBLK Type* field identifies the type of DBLK (MTF_SSET, MTF_VOLB, etc.). Developers may add their own DBLK types but they must be approved by the MTF Review Committee prior to implementation. Application software must be able to handle the presence of unknown DBLKs. If an unknown DBLK type is encountered, the application software should use the information in the MTF_DB_HDR section to skip the information associated with the DBLK.

Note: The values of the IDs were selected such that when viewed as a hex dump, they are easily identifiable in the ASCII portion and match the names used for them up till this point.

Table 2. Block ID Table

DBLK Name	Description	Four Character ID	Hex Value
MTF_TAPE	**TAPE** descriptor block	'TAPE'	0x45504154
MTF_SSET	**S**tart of data **SET** descriptor block	'SSET'	0x54455353
MTF_VOLB	**VOL**ume descriptor **B**lock	'VOLB'	0x424C4F56
MTF_DIRB	**DIR**ectory descriptor **B**lock	'DIRB'	0x42524944
MTF_FILE	**FILE** descriptor block	'FILE'	0x454C4946
MTF_CFIL	Corrupt object descriptor block	'CFIL'	0x4C494643
MTF_ESPB	**E**nd of **S**et **P**ad descriptor **B**lock	'ESPB'	0x42505345
MTF_ESET	**E**nd of **SET** descriptor block	'ESET'	0x54455345
MTF_EOTM	**E**nd **O**f **T**ape **M**arker descriptor block	'EOTM'	0x4D544F45
MTF_SFMB	**S**oft **F**ile**M**ark descriptor **B**lock	'SFMB'	0x424D4653

Block Attributes {4 bytes}

The *Block Attributes* is a 32-bit field used to specify the attributes of a DBLK. The defined bit values for this field are shown in the table below and discussed in the following table. These attribute bits are directly related to tape format issues.

The least significant 16-bits (BIT0 - BIT15) are valid for any DBLK and the most significant 16-bits (BIT16 - BIT31) are valid only for the specific DBLK listed in the table. This method allows for multiple definitions for the same bit depending on the DBLK context. Note that those bits listed as valid in *any* DBLK may not be valid in *all* DBLKs, but are used in more than one type. Missing bits in the 32-bit field are reserved for future use.

Table 3. Block Attributes (MTF_DB_HDR)

Name	Description	DBLK Type	Value
MTF_CONTINUATION	Bit set if DBLK is a continuation from the previous tape.	any	BIT0
MTF_COMPRESSION	Bit set if compression *may* be active.	any	BIT2
MTF_EOS_AT_EOM	Bit set if the End Of Medium was hit during end of set processing.	any	BIT3
MTF_SET_MAP_EXISTS	Bit set if an Media Based Catalog Set Map can be found on the tape.	MTF_TAPE	BIT16
MTF_FDD_ALLOWED	Bit set if an attempt will be made to put a Media Based Catalog File/Directory Detail section on the tape.	MTF_TAPE	BIT17
MTF_FDD_EXISTS	Bit set if a Media Based Catalog File/Directory Detail section has been successfully put on the tape for this Data Set.	MTF_SSET	BIT16
MTF_ENCRYPTION	Bit set if encryption is active for the data streams within this Data Set.	MTF_SSET	BIT17
MTF_FDD_ABORTED	Bit set if a Media Based Catalog File/Directory Detail section was aborted for any reason during the write operation.	MTF_ESET	BIT16
MTF_END_OF_FAMILY	Bit set if the Media Based Catalog Set Map has been aborted. This condition means that additional Data Sets cannot be appended to the tape.	MTF_ESET	BIT17
MTF_ABORTED_SET	Bit set if the Data Set was aborted while being written. This can happen if a fatal error occurs while writing data, or if the user terminates the data management operation. An MTF_ESET DBLK containing this flag is put at the end of the Data Set even if it was aborted.	MTF_ESET	BIT18
MTF_NO_ESET_PBA	Bit set if no Data Set ends on this tape (i.e. continuation tape must follow this tape).	MTF_EOTM	BIT16
MTF_INVALID_ESET_PBA	Bit set if the Physical Block Address (PBA) of the MTF_ESET is invalid because the tape drive doesn't support physical block addressing.	MTF_EOTM	BIT17

Note: BIT0 - BIT31 represent the individual bits of a 32-bit value. BIT0 is the least significant bit and BIT31 is the most significant bit.

Offset To First Event {2 bytes}

The *Offset To First Event* field is used as an offset from the start of the DBLK to the first data stream associated with the DBLK. If there are no data streams associated with a DBLK, then this field contains the offset to the next DBLK.

Note: This is used for backwards compatibility with earlier drafts of the MTF Version 1.0 specification. MTF Version 1.00a specifies that all DBLKs have at least one data stream associated with it and that the last data stream be the SPAD data stream.

OS ID {2 bytes}

The *OS ID* field identifies the operating system associated with the information in this DBLK. Values currently defined for this field and for the *OS Version* field are listed in Appendix A Operating System Specific Data. Developers may add new values for this field, but new values must be "registered".

OS Version {2 bytes}

The *OS Version* field identifies the version of the operating system specified in the *OS ID* field. The "version" specified here is not a release version of an operating system (as in Windows NT Version 3.5) but rather the version of a structure for representing OS specific information within DBLKs. See Appendix A Operating System Specific Data for more information.

Displayable Size {8 bytes}

The *Displayable Size* field uses the UINT64 low level structure to specify the size that may be displayed by an application for this DBLK. For example, the size of a file would be stored here for MTF_FILE DBLKs. The size displayed to a user may be different from the physical size of an object and therefore should be used for display purposes only.

Format Logical Address {8 bytes}

The *Format Logical Address* field also uses the UINT64 low level structure to specify the Format Logical Address (number of Format Logical Blocks from the first MTF_SSET in this Data Set) of this DBLK. Refer to Section 3.7 for more information on Format Logical Addresses.

Reserved for MBC {2 bytes}

The *Reserved for Media Based Catalog (MBC)* field is used to store application specific information in the Type 2 MBC-SLO Set Map and FDD. This field is set to zero outside of the Type 2 MBC-SLO Set Map and FDD.

Control Block ID {4 bytes}

The *Control Block ID* field is used for error recovery. The MTF_SSET DBLK has a Control Block ID value of zero. All subsequent DBLKs within the Data Set will have a Control Block ID one greater than the previous DBLK's Control Block ID. Values for this field are only defined for DBLKs within a Data Set from the MTF_SSET to the last DBLK occurring prior to the MTF_ESET.

OS Specific Data {4 bytes}

The *OS Specific Data* field uses an MTF_TAPE_ADDRESS low level structure to identify the location and size of an OS specific structure. The contents of the structure are dependent upon the values of the *OS ID* and *OS Version* fields as well as the type of DBLK. The structures for the identified operating systems and versions are defined in Appendix C.

String Type {1 byte}

The *String Type* is a single byte field that specifies the format of strings stored in this DBLK. The table below specifies acceptable values for this field.

Table 4. String Types

Name	Description	Value
NO_STRINGS	Indicates there are no strings associated with the DBLK.	0
ANSI_STR	Indicates that strings are single byte ANSI code.	1
UNICODE_STR	Indicates that strings are two byte Unicode.	2

Header Checksum {2 bytes}

The *Header Checksum* field is a 16-bit word-wise XOR sum of all the fields of the MTF_DB_HDR except for the checksum field itself. This field may be used to detect data corruption on tape.

Reserved

Reserved fields should not be used to store information as they are reserved for future use. Reserved fields should be zero filled.

5.1.1 DBLK Specific Attribute Bits

In addition to the attribute field in the DBLK Header, there is an attribute field in the block specific section of all DBLKs except the MTF_EOTM and MTF_ESPB. These attributes pertain to the content of the data, rather than its layout on tape. For example, there are bits in the attributes field of the MTF_SSET DBLK indicating what type of data management operation (transfer, copy, normal backup, incremental backup, etc.) was used to create the Data Set. There are bits in the attributes field of the MTF_DIRB and MTF_FILE DBLKs that indicate whether the directory or file represented by the DBLK is read only, hidden, system or has been modified since the last backup.

Definitions of DBLK Specific Attribute Bits and their use can be found in the individual DBLK descriptions that follow. Not all DBLKs containing this field have bits defined for them at this time. The field was added in anticipation of future use.

There is one thing which all the DBLK specific attribute fields have in common. The high byte of these 32-bit fields (BIT24 - 31) is available for *vendor specific* attributes. These bits do not have to be registered. It is important to note that, in order to preserve data interchangeability, the vendor specific bits **should not** contain information required to properly restore the data.

5.1.2 Strings Within DBLKs

The length of strings within DBLKs is determined by the *Size* field in the MTF_TAPE_ADDRESS low level structure that refers to them. Unless otherwise noted, these strings are not NULL terminated and are of the string type specified for that DBLK in the *String Type* field of the MTF_DB_HDR.

5.2 DBLK Structures

Descriptor Block structures (DBLKs) are the basic structural components of the Microsoft Tape Format. DBLKs are headers that provide information necessary to locate and interpret the data on the tape. These DBLKs contain fields, some of which are low level structures, to describe and identify tapes, Data Sets, and the individual objects (e.g. volume, directories, files, etc.) that comprise Data Sets.

All DBLKs defined in MTF include the 52 byte Common Block Header (MTF_DB_HDR) structure at the head of the DBLK structure. The MTF_DB_HDR is typically followed by additional fields and sometimes by an OS Specific Data area and String Storage area. The maximum length of a DBLK in MTF is 1024 bytes.

5.2.1 Tape Header Descriptor Block (MTF_TAPE)

The **Tape Header Descriptor Block** (MTF_TAPE DBLK) contains general information that applies to the current media. The MTF_TAPE DBLK is the first block on a media and contains information that is crucial to media families, such as the media sequence, ID, name, description, etc. Other fields in the MTF_TAPE DBLK identify characteristics of the media that must remain constant. These characteristics can include the MTF major revision, the Media Based Catalog type used, and the Format Logical Block size which specifies the byte alignment of all DBLKs written to the media.

Offset	Field Name	Type	Size
0 *0h*	Common Block Header	MTF_DB_HDR	52 bytes
52 *34*h	Media Family ID	UINT32	4 bytes
56 *38h*	TAPE Attributes	UINT32	4 bytes
60 *3Ch*	Media Sequence Number	UINT16	2 bytes
62 *3Eh*	Password Encryption Algorithm	UINT16	2 bytes
64 *40h*	Soft Filemark Block Size	UINT16	2 bytes
66 *42h*	Media Based Catalog Type	UINT16	2 bytes
68 *44h*	Media Name	MTF_TAPE_ADDRESS	4 bytes
72 *48h*	Media Description/Media Label	MTF_TAPE_ADDRESS	4 bytes
76 *4Ch*	Media Password	MTF_TAPE_ADDRESS	4 bytes
80 *50h*	Software Name	MTF_TAPE_ADDRESS	4 bytes
84 *54h*	Format Logical Block Size	UINT16	2 bytes
86 *56h*	Software Vendor ID	UINT16	2 bytes
88 *58h*	Media Date	MTF_DATE_TIME	5 bytes
93 *5Dh*	MTF Major Version	UINT8	1 byte

Structure 5. Tape Header Descriptor Block (MTF_TAPE)

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The following fields of the MTF_DB_HDR structure must be set to the defined value.

- The *DBLK Type* field is set to 'TAPE'.
- The *Format Logical Address* field is set to zero.
- The *Control Block ID* field is set to zero.

Media Family ID {4 bytes}

The *Media Family ID* field is a four byte number that identifies the Media Family to which this media belongs. A continuation media must have the same Media Family ID as the previous media. MTF Version 1.00a does not specify an algorithm for generating unique Media Family ID numbers.

TAPE Attributes {4 bytes}

The *TAPE Attributes* field is a four byte (32-bit) field specifying attributes that pertain to the content of data on this media. Bits 0 -1 are defined below. Bits 2 - 23 are reserved for future use, and the most significant 8-bits (BIT24 - BIT31) are reserved for vendor specific attributes.

Table 5. TAPE Attributes

Name	Description	Value
TAPE_SOFT_FILEMARK_BIT	This bit is set if the soft filemarks are being used. The Soft Filemark Block Size field must be set.	BIT0
TAPE_MEDIA_LABEL_BIT	This bit is set if the Media Description/Media Label field contains a Media Label.	BIT1
	Reserved (set to zero)	BIT2 - BIT23
	Vendor Specific	BIT24 - BIT31

Media Sequence Number {2 bytes}

The *Media Sequence Number* field will start at "1" with the first media, and will increment by one for each new media processed in a Media Family.

Password Encryption Algorithm {2 bytes}

The *Password Encryption Algorithm* field indicates the algorithm used to encrypt the password data associated with the *Media Password* field. Applications should not access media if a password exists and the encryption algorithm is unknown. This value must be a "registered" encryption algorithm number.

Soft Filemark Block Size {2 bytes}

The *Soft Filemark Block Size* field contains the size of the Soft Filemark (MTF_SFMB) DBLK in multiples of 512 bytes (e.g., a value of 2 equals a MTF_SFMB DBLK size of 1024 bytes). The Soft Filemark Block Size is calculated from the physical block size as reported by the device driver. The Soft Filemark Block Size is only required for Soft Filemark emulation.

Media Based Catalog Type {2 bytes}

The *Media Based Catalog Type* field indicates which of the formats described in Appendix B is used for Media Based Catalogs. The "type" of Media Based Catalog (MBC) must remain consistent across an entire Media Family and is therefore identified here in the MTF_TAPE DBLK.

There is an MBC "version" defined in the MTF_SSET DBLK which identifies minor version changes for a specific MBC type.

Table 6. Media Based Catalog Types

Name	Value
No MBC used	0
Type 1 MBC	1
Type 2 MBC	2

Media Name {4 bytes}

The *Media Name* field uses the four byte MTF_TAPE_ADDRESS low level structure to specify the location and size of a string used to identify the media to a user. If no name is associated with the media, then the *Size* field in the MTF_TAPE_ADDRESS low level structure will be zero.

Media Description/Media Label {4 bytes}

The *Media Description/Media Label* field also uses the four byte MTF_TAPE_ADDRESS low level structure to specify the location and size of a Media Description string or a software generated Media Label. If a Media Label is used, the TAPE_MEDIA_LABEL_BIT of the TAPE Attributes filed is set. If no Description/Label is associated with the media then the *Size* field in the MTF_TAPE_ADDRESS low level structure will be zero.

Media Description Definition

A Media Description is used to describe the contents of the media in a human readable form

Media Label Definition

A Media Label is a unique identifier that is generated by an application when a MTF or non MTF media is introduced. The Media Label is used by the application for media management. Once a unique Media Label is generated by an application, the application must guarantee that the Media Label is preserved each time the media is overwritten. A Media Label is comprised of the Tag, Version, Vendor, Vendor ID, Creation Time Stamp, Cartridge Label, Side, Media ID, Media Domain ID, and Vendor Specific fields. All fields in the Media Label are separated by the '|' character and are of the alpha numeric type with the inclusion of the following characters '+', '-', '_', ':', '/', '.', '{', and '}'.

Table 7. Media Label

Name	Description
Tag	The tag field identifies this as a Media Label. This field is set to the string "MTF Media Label".
Version	The Media Label version number. This is set to the three characters '1.0'.
Vendor	The name of the vendor that created the Media Label.
Vendor Product ID	The vendor product ID field is used to uniquely identify the product that generated this Media Label. The vendor product ID is determined by the vendor. This field is optional. If unused this field is empty.
Creation Time Stamp	Date and time the Media Label was originally generated. The creation time stamp is in the YYYY/MM/DD.HH:MM:SS format.
Cartridge Label	The identifier that is printed on the label that is affixed to the cartridge or the bar code label.
Side	The side field contains the current side number of the media. For single sided media, this field is always set to the character '1'. For double sided media, this field is set to the character '1' on the primary side and the character '2' on the opposite side. NOTE: On double sided media, the Tag, Version, Vendor, Vendor Product ID, Creation Time Stamp, Cartridge Label, Media ID, Media Domain ID, and Media Domain Name fields must be identical on both sides.
Media ID	The media ID is a globally unique identifier—128-bit integer that is guaranteed to be unique in the world across space and time. This globally unique identifier is also known as a UUID (universally unique ID) as defined by the Open Software Foundation's Distributed Computing Environment. If this field was not generated using the specified UUID algorithm, it cannot start with the character '{'.
Media Domain ID	The media domain ID is also a UUID. It is used to identify the media domain in which the media was labeled. A domain is a vendor specific collection of resources (e.g., a backup server). If this field was not generated using the specified UUID algorithm, it cannot start with the character '{'. This field is optional. If unused this field is empty.

Vendor Specific	Vendor specific extensions to the Media Label. Vendor specific extensions are optional. All vendor specific extensions start with the three characters 'VS:'.

Example Media Label:

MTF Media Label|1.0|Seagate|PVL|1996/03/29.18:36:10|AB1234|1|{9EAA3460-89BA-11cf-8A04-0000C0D9CA0D}|{7E43CEA0-89BA-11cf-8A04-0000C0D9CA 0D}| VS:First Vendor Specific Parameter

Media Password {4 bytes}

The *Media Password* field uses the MTF_TAPE_ADDRESS low level structure to specify the location and size of a string containing the password for this media. The associated data will be encrypted using the algorithm specified by the *Password Encryption Algorithm* field. If no password is associated with the media, then the *Size* field in the MTF_TAPE_ADDRESS structure will be zero.

Software Name {4 bytes}

The *Software Name* field is an MTF_TAPE_ADDRESS low level structure specifying the location and size of a string containing the name of the software application that created this media. The *Size* field of the MTF_TAPE_ADDRESS structure should never be zero.

Format Logical Block Size {2 bytes}

The *Format Logical Block Size* is a two byte field used to specify the alignment for all DBLKs on the media. Valid values for the *Format Logical Block Size* are 512 and 1024 bytes.

Software Vendor ID {2 bytes}

The *Software Vendor ID* is a two byte field that identifies the vendor ID of the software application that wrote this media. This value must be a "registered" software vendor ID number.

Media Date {5 bytes}

The *Media Date* field uses the five byte MTF_DATE_TIME low level structure which indicates the exact date and time that this media was first created. The resolution is down to the second.

MTF Major Version {1 byte}

The *MTF Major Version* is a single byte field used to identify the major version of Microsoft Tape Format used to create this media. Version numbers start at "1" as in MTF Version 1.00a and increment by one with each major revision. This field allows for 255 major versions. All sets written to this Media Family must use the same major version of this format.

An MTF Minor Version is identified in the MTF_SSET DBLK. MTF Version 1.00a has a minor version number of "0". Minor versions of an MTF Major Version may have differences in the structure of DBLK fields but must maintain backwards compatibility (i.e. fields cannot be removed, only added). Please refer to the MTF_SSET DBLK description for more information on this subject.

Table 8. Major Version Numbers

Name	Value
MTF Major Version	1

5.2.2 Start of Data Set Descriptor Block (MTF_SSET)

The **Start of Data Set Descriptor Block** (MTF_SSET DBLK) contains information describing the Data Set. The MTF_SSET DBLK is put at the beginning of an entire Data Set. This structure contains information identifying and describing all important aspects of the Data Set such as: the Physical Block Address (PBA), user name and password, data management software version, encryption and compression algorithms used, Data Set number, backup attributes, media write time, time zone, MTF minor version used, etc.

The organization of the MTF_SSET structure uses a MTF_DB_HDR structure followed by a number of fields and a String Storage Area. The String Storage Area is used for storing strings such as the name of the Data Set, the user name, etc.

Offset	Field Name	Type	Size
0 ***0h***	Common Block Header	MTF_DB_HDR	52 bytes
52 ***34h***	SSET Attributes	UINT32	4 bytes
56 ***38h***	Password Encryption Algorithm	UINT16	2 bytes
58 ***3Ah***	Software Compression Algorithm	UINT16	2 bytes
60 ***3Ch***	Software Vendor ID	UINT16	2 bytes
62 ***3Eh***	Data Set Number	UINT16	2 bytes
64 ***40h***	Data Set Name	MTF_TAPE_ADDRESS	4 bytes
68 ***44h***	Data Set Description	MTF_TAPE_ADDRESS	4 bytes
72 ***48h***	Data Set Password	MTF_TAPE_ADDRESS	4 bytes
76 ***4Ch***	User Name	MTF_TAPE_ADDRESS	4 bytes
80 ***50h***	Physical Block Address (PBA)	UINT64	8 bytes
88 ***58h***	Media Write Date	MTF_DATE_TIME	5 bytes
93 ***5Dh***	Software Major Version	UINT8	1 byte
94 ***5Eh***	Software Minor Version	UINT8	1 byte
95 ***5Fh***	Time Zone	INT8	1 byte
96 ***60h***	MTF Minor Version	UINT8	1 byte
97 ***61h***	Media Catalog Version	UINT8	1 byte

Structure 6. Start of Set Descriptor Block (MTF_SSET)

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The *DBLK Type* field within the MTF_DB_HDR will be set to 'SSET'.

SSET Attributes {4 bytes}

The *SSET Attributes* field is four bytes in length organized as a 32-bit field. Only Bits 0 - 5 are defined at this time. Bits 1 - 5 are used to specify what type of backup operation was used to create the Data Set immediately following the MTF_SSET DBLK on the media. Possible operation types include, copy, normal backup, differential backup, incremental backup and daily backup. Only one of these five bits should be set for a given Data Set. In the descriptions that follow, the "modified" flag (describing whether a file has been created or modified) is mentioned. Another name for this is the "archive" flag. Bits 6 - 23 of this field are reserved for future use.

Table 9. SSET Attributes

Name	Description	Value
SSET_TRANSFER_BIT	This bit is set if the data management operation is a “transfer”. It indicates that the files in this Data Set were removed from the source media after the operation was completed.	BIT0
SSET_COPY_BIT	This bit is set if the operation is a “copy”. The copy method copies all selected files from the primary storage to the media. The file’s “modified” flag IS NOT reset afterwards.	BIT1
SSET_NORMAL_BIT	This bit is set if the backup type is “normal”. The normal backup method backs up all selected files. The file’s “modified” flag IS reset afterwards.	BIT2
SSET_DIFFERENTIAL_BIT	This bit is set if the backup type is “differential”. The differential backup method only backs up selected files having their “modified” flag set. The file’s “modified” flag IS NOT reset afterwards.	BIT3
SSET_INCREMENTAL_BIT	This bit is set if the backup type is “incremental”. The incremental backup method only backs up selected files having their “modified” flag set. The file’s “modified” flag IS reset afterwards.	BIT4
SSET_DAILY_BIT	This bit is set if the backup type is “daily”. The daily backup method only backs up selected files created or modified with today’s date. The file’s “modified” flag IS NOT reset afterwards.	BIT5
	Reserved (set to zero)	BIT6 - BIT23
	Vendor Specific	BIT24 - BIT31

Password Encryption Algorithm {2 bytes}

The *Password Encryption Algorithm* field is a two byte field indicating the ID of the algorithm used to encrypt the *Data Set Password* field. Applications should not access media if a password exists and the encryption algorithm is unknown. This value must be a "registered" encryption algorithm number.

Software Compression Algorithm {2 bytes}

The *Software Compression Algorithm* field is also a two byte field indicating the ID of the algorithm used to compress the data streams associated with all DBLKs within the Data Set. This 2 byte value must be a "registered" compression algorithm number.

Note: If the MTF_COMPRESSION bit in the Block Attributes field of the Common Block Header is set and the Software Compression Algorithm field is zero, the software compression algorithm used cannot be determined until the first compressed Stream Header is encountered.

Software Vendor ID {2 bytes}

The *Software Vendor ID* field identifies the vendor of the software that wrote this Data Set. This value must be a "registered" software vendor ID number.

Data Set Number {2 bytes}

The *Data Set Number* field is a two byte field containing the ID number corresponding to this Data Set. Data Set Numbers start at “one” (0x01) with the first Data Set in the Media Family, and are incremented by one for each new Data Set appended to the Media Family. When a Data Set continues on a new media, the MTF_CONTINUATION bit in the *Block Attributes* field of the MTF_DB_HDR structure of the MTF_SSET DBLK will be set, but the Data Set Number will remain the same.

Data Set Name {4 bytes}

The *Data Set Name* field uses the 4 byte MTF_TAPE_ADDRESS low level structure to specify the size and location of a string. This string, located in the String Storage Area, contains the user name given to the Data Set. If no name is associated with the Data Set, then the *Size* field of the MTF_TAPE_ADDRESS structure will be zero.

Data Set Description {4 bytes}

The *Data Set Description* field uses the 4 byte MTF_TAPE_ADDRESS low level structure to specify the size and location of a string containing a description of the Data Set for the user. If no description is associated with the Data Set, then the *Size* field in the MTF_TAPE_ADDRESS structure will be zero.

Data Set Password {4 bytes}

The *Data Set Password* field uses the MTF_TAPE_ADDRESS low level structure to specify the size and location of a string containing the password for this Data Set. The associated data in the string will be encrypted using the algorithm specified by the *Password Encryption Algorithm* field. If no password is associated with the Data Set, then the *Size* field in the MTF_TAPE_ADDRESS structure will be zero.

User Name {4 bytes}

The *User Name* field uses the MTF_TAPE_ADDRESS low level structure to specify the size and location of a string indicating the user name of the account that generated this Data Set. If no user name is associated with the Data Set, then the Size field in the MTF_TAPE_ADDRESS structure will be zero.

Physical Block Address (PBA) {8 bytes}

The *Physical Block Address (PBA)* field uses a UINT64 structure to specify the Physical Block Address of this MTF_SSET DBLK. PBAs are obtained from the device. It is critical that all software and/or device drivers conform to the same rules for generating PBAs. Refer to section 3.6 for a general description of PBAs, and to Appendix L for device specific details.

Media Write Date {5 bytes}

The *Media Write Date* field uses the five byte MTF_DATE_TIME low level structure to indicate the exact date and time that this Data Set was created.

Software Major Version {1 byte}

The *Software Major Version* is a one byte field used to specify the major version number of the software application used to create this Data Set. For instance, this field would contain the integer "3" if *Ultimate Enterprise Backup Ver. 3.1* was used to create the Data Set.

Software Minor Version {1 byte}

The *Software Minor Version* is another 1 byte field which specifies the minor version number of the software application used to create the Data Set. This field would contain the value "1" using the example described above. The major and minor software versions are vendor specific and must have integer values in the range of 0 to 255.

Time Zone {1 byte}

The *Time Zone* field is a single byte that indicates the difference between the local time and UCT. This difference is stored as the number of fifteen minute intervals between the two time zones. Therefore, the value of this field must be between -48 and +48. (i.e. EST is -20 since it is five hours after UCT). If time values are not coordinated with UCT, then this field must be set to "127".

Table 10. Time Zones

Name	Description	Value
LOCAL_TZ	Indicates that local time is not coordinated with UCT.	127

MTF Minor Version {1 byte}

The *MTF Minor Version* field is a single byte field indicating the minor version of a major MTF version. For example, MTF Version 2.3 would have a minor version of "3". The minor version can vary from one Data Set to another and is therefore identified in the MTF_SSET DBLK, whereas the major version must remain consistent for an entire media and is therefore identified in the MTF_TAPE DBLK.

If additional fields are added to a DBLK, then the MTF minor version will be increased. No fields will be deleted in any later minor versions of the format. Therefore, if this number is greater than or equal to the minor version of the format that your software understands, then your program can expect all fields to be properly initialized.

Minor versions start at "0" for all major versions of MTF. This field allows for 255 minor versions.

Table 11. Minor Version Numbers

Name	Value
MTF Minor Version for Major Version "1"	0

Refer to the MTF_TAPE DBLK structure where the *MTF Major Version* is defined.

Media Catalog Version {1 byte}

The *Media Catalog Version* field is a single byte field indicating the version of the Media Based Catalog written to this media. A Media Based Catalog has a "type" and a "version". This is similar in effect to the major and minor versions of the MTF format. The version can be changed on a Data Set basis and is therefore identified in the MTF_SSET DBLK. For instance, different versions of a Media Based Catalog may have differences in the fields of the File/Directory Detail. The type of MBC cannot change within a Media Family and is therefore identified in the MTF_TAPE DBLK. Values for the *Media Catalog Version* field are specified in the Media Based Catalog definitions.

5.2.3 Volume Descriptor Block (MTF_VOLB)

The **Volume Descriptor Block** (MTF_VOLB DBLK) contains information describing a source volume in the Data Set. The MTF_VOLB DBLK structure contains physical volume information describing the physical location of the file(s) being written or read. It consists of a MTF_DB_HDR followed by an area with fields specific to the MTF_VOLB DBLK, a String Storage Area used for storing the device, volume and machine names. String storage can be located anywhere following the defined fields.

When writing the files a user sees on a network, these files may appear to be on drives E:, F: and G:, but in reality some may be on a local drive and others may be on one or more network servers. The MTF_VOLB DBLK will reference data that describes the physical location of the files based on the user's logical view at the time of the data management operation. A MTF_VOLB DBLK must precedes any Directory Descriptor Blocks (MTF_DIRB DBLKs) for a given volume.

Offset	Field Name	Type	Size
0 ***0h***	Common Block Header	MTF_DB_HDR	52 bytes
52 ***34h***	VOLB Attributes	UINT32	4 bytes
56 ***38h***	Device Name	MTF_TAPE_ADDRESS	4 bytes
60 ***3Ch***	Volume Name	MTF_TAPE_ADDRESS	4 bytes
64 ***40h***	Machine Name	MTF_TAPE_ADDRESS	4 bytes
68 ***44h***	Media Write Date	MTF_DATE_TIME	5 bytes

Structure 7. Volume Descriptor Block (MTF_VOLB)

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The *DBLK Type* field within the MTF_DB_HDR will be set to 'VOLB'. The MTF_DB_HDR contains a number of fields common to all DBLKs as well as an offset field for locating the OS Specific Data area. This area contains directory related information for a specific operating system.

VOLB Attributes {4 bytes}

The *VOLB Attributes* field is four bytes in length organized as a 32-bit field. VOLB Attributes define characteristics of the Volume Block represented by this MTF_VOLB DBLK. Only Bits 0 - 5 are defined at this time. Bits 6 - 23 are reserved for future use. Bits 2 - 5 define the format of the device name. Exactly one of these bits must be set in all MTF_VOLB DBLKs.

Table 12. VOLB Attributes

Name	Description	Value
VOLB_NO_REDIRECT_RESTORE_BIT	Objects following this DBLK can only be restored to the device from which they were backed up.	BIT0
VOLB_NON_VOLUME_BIT	Objects following this DBLK are not associated with a volume.	BIT1
VOLB_DEV_DRIVE_BIT	Device name format is, "<drive letter>:".	BIT2
VOLB_DEV_UNC_BIT	Device name format is UNC.	BIT3
VOLB_DEV_OS_SPEC_BIT	Device name format is OS specific (refer to Appendix C for details on a given OS).	BIT4
VOLB_DEV_VEND_SPEC_BIT	Device name format is vendor specific.	BIT5
	Reserved (set to zero)	BIT6 - BIT23
	Vendor Specific	BIT24 - BIT31

Note: In cases where the data objects are not associated with a volume, the MTF_VOLB DBLK is still needed to store the device and machine names.

Device Name {4 bytes}

The *Device Name* field is a four byte MTF_TAPE_ADDRESS low level structure used to specify a length and offset to a string in the String Storage area. The string referred to by this field identifies the actual backup source name, which is used as the default restore target (potentially the only allowable target if redirection of the data contained in the set is not permitted).

Most current applications use drive letters (C:, D:, E:, etc.) or UNC names in the device name field, which are very portable, but others such as NetWare SMS use specialized, OS specific device names which are not portable. Refer to Appendix C for OS specific information on device name formats. Vendors may also choose to define their own device name format to meet special requirements of their application. However, it should be noted that a vendor specific device name for a volume which cannot be redirected would require the user to have knowledge of the source device name when restoring such a set with a different application. In all cases, one of the VOLB attribute bits (BIT2-BIT5) defined above should be set to define the device name format being used.

Volume Name {4 bytes}

The *Volume Name* field is another four byte MTF_TAPE_ADDRESS low level structure used to specify a length and offset to the string that identifies the name associated with the volume (i.e. SYS, MAIL, etc.). The string referred to by this field is for display purposes only. It is used to store things such as volume labels and network share comments.

Machine Name {4 bytes}

The *Machine Name* field is another four byte MTF_TAPE_ADDRESS low level structure used to indicate the size and offset of the string containing the name of the machine that this volume is on (i.e. PENTIUM_1, ENG_SERV, etc.).

Media Write Date {5 bytes}

The *Media Write Date* field is a five byte MTF_DATE_TIME low level structure containing the date and time that the media was first written for this volume.

5.2.4 Directory Descriptor Block (MTF_DIRB)

The **Directory Descriptor Block** (MTF_DIRB DBLK) contains the information required for restoring a directory. It consists of a MTF_DB_HDR followed by an area with fields specific to the MTF_DIRB DBLK, an OS Specific Data Area and a String Storage Area used for storing the file name. The area preceding the OS Specific Data and String Storage Areas contains fields of information about the file that is valid across platforms. The OS Specific Data Area and String Storage Area can be placed anywhere following the MTF_DIRB DBLK specific area and can be reversed in order. MTF_DIRB DBLK(s) must precedes any File Descriptor Blocks (MTF_FILE DBLKs) for a given directory.

Offset	Field Name	Type	Size
0 ***0h***	Common Block Header	MTF_DB_HDR	52 bytes
52 ***34h***	DIRB Attributes	UINT32	4 bytes
56 ***38h***	Last Modification Date	MTF_DATE_TIME	5 bytes
61 ***3Dh***	Creation Date	MTF_DATE_TIME	5 bytes
66 ***42h***	Backup Date	MTF_DATE_TIME	5 bytes
71 ***47h***	Last Access Date	MTF_DATE_TIME	5 bytes
76 ***4Ch***	Directory ID	UINT32	4 bytes
80 ***50h***	Directory Name	MTF_TAPE_ADDRESS	4 bytes

Structure 8. Directory Descriptor Block (MTF_DIRB)

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The *DBLK Type* field within the MTF_DB_HDR will be set to 'DIRB'. The MTF_DB_HDR contains a number of fields common to all DBLKs, as well as an offset field used for locating the OS Specific Data Area. This area contains directory related information for a specific operating system.

DIRB Attributes {4 bytes}

The *DIRB Attributes* field is four bytes in length organized as a 32-bit field. DIRB Attributes define characteristics of the directory represented by this MTF_DIRB DBLK. Bits 0 - 7 are reserved for future use. Bits 8 - 11 and 16 - 18 are described in the table below. Other bits through bit 23 are reserved for future use. These bits describe directory attributes common to most operating systems. Some operating systems do not make use of some of these attributes and where that is the case, these bits can simply be ignored.

Table 13. DIRB Attributes

Name	Description	Value
DIRB_READ_ONLY_BIT	This bit is set if the directory is marked as read only.	BIT8
DIRB_HIDDEN_BIT	This bit is set if the directory is hidden from the user.	BIT9
DIRB_SYSTEM_BIT	This bit is set if the directory is a system directory.	BIT10
DIRB_MODIFIED_BIT	This bit is set if the directory has been modified. This is also referred to as an "archive" flag.	BIT11
DIRB_EMPTY_BIT	This bit set if the directory contained no files or subdirectories.	BIT16
DIRB_PATH_IN_STREAM_BIT	This bit set if the directory path is stored in a stream associated with this DBLK.	BIT17
DIRE_CORRUPT_BIT	This bit set if the data associated with the directory could not be read.	BIT18
	Reserved (set to zero)	BIT0 - BIT7 BIT12 - BIT15 BIT19 - BIT23
	Vendor Specific	BIT24 - BIT31

Last Modification Date {5 bytes}

The *Last Modification Date* field is five bytes in length and contains the date and time that the directory was last modified. A directory is considered modified whenever a file or directory belonging to this parent directory is added or removed. This field uses a MTF_DATE_TIME low level structure.

Creation Date {5 bytes}

The *Creation Date* field is another five byte field using the MTF_DATE_TIME low level structure. This field contains the date and time when the directory was first created.

Backup Date {5 bytes}

The *Backup Date* field is another five byte MTF_DATE_TIME field containing the date and time that the directory was last backed up. A directory is considered "backed up" when its entire contents are backed up.

Last Access Date {5 bytes}

The *Last Access Date* field also uses the five byte MTF_DATE_TIME low level structure to describe the date and time that the directory was last accessed. A directory is considered "accessed" when its contents are modified in any way.

Note: Backup programs should not affect this field if possible.

Directory ID {4 bytes}

The *Directory ID* is a four byte field containing the ID of the directory. This ID starts at one for the first directory in a Data Set and is incremented by one for each additional directory processed. This field is used for error handling and recovery.

Directory Name {4 bytes}

The *Directory Name* field is four bytes in length using an MTF_TAPE_ADDRESS low level structure that specifies the location and size of the name associated with this directory. The directory name does not include the server, volume or drive. In addition, the “root” indicator ‘\’ must not be the first character. It is assumed that all directories start from the root. The path separator for the native system must be replaced by a NULL character ('\0'). The *Size* field within the MTF_TAPE_ADDRESS must be used to determine the length of the name string.

The “root" directory would be stored as a one character length string with a single NULL character. For consistency with the specification of the "root", all directory names are stored with a trailing NULL character. Note that this is a trailing path separator, **not** a string terminator.

The entry for a "root" directory (i.e. "C:\") is stored in the DBLK as:

```
'\0'
```

The entry for the directory "C:\apps\fred\bloggs\" is stored in the DBLK as:

```
apps'\0'fred'\0'bloggs'\0'
```

Since the size of the path may result in a DBLK size that is larger than the maximum allowed, larger directory names would have to be stored in a separate 'PNAM' data stream. This stream must be the first data stream following the MTF_DIRB DBLK. When restoring this directory to a different operating system, the name used to create the directory may have to be modified to eliminate characters that are not valid for the target operating system or to adjust the size to the maximum directory name length for the target operating system.

5.2.5 File Descriptor Block (MTF_FILE)

The **File Descriptor Block** (MTF_FILE DBLK) contains the information required for restoring a file. It consists of a MTF_DB_HDR followed by an area with fields specific to the MTF_FILE DBLK, an OS Specific Data area and a String Storage Area used for storing the file name. The area preceding the OS Specific Data and String Storage Areas contains fields of information about the file that is valid across platforms. The OS Specific Data and String Storage Areas can be placed anywhere following the MTF_FILE DBLK specific area.

Offset	Field Name	Type	Size
0 ***0h***	Common Block Header	MTF_DB_HDR	52 bytes
52 ***34h***	FILE Attributes	UINT32	4 bytes
56 ***38h***	Last Modification Date	MTF_DATE_TIME	5 bytes
61 ***3Dh***	Creation Date	MTF_DATE_TIME	5 bytes
66 ***42h***	Backup Date	MTF_DATE_TIME	5 bytes
71 ***47h***	Last Access Date	MTF_DATE_TIME	5 bytes
76 ***4Ch***	Directory ID	UINT32	4 bytes
80 ***50h***	File ID	UINT32	4 bytes
84 ***54h***	File Name	MTF_TAPE_ADDRESS	4 bytes

Structure 9. File Descriptor Block (MTF_FILE)

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The *DBLK Type* field within MTF_DB_HDR will be set to 'FILE'. The MTF_DB_HDR contains a number of fields common to all DBLKs as well as an offset field for locating the OS Specific Data Area.

FILE Attributes {4 bytes}

The *FILE Attributes* field is four bytes in length organized as a 32-bit field. This field specifies the attributes of the file represented by this DBLK.

The table below shows the bits currently defined and those reserved for future use. These bits describe file attributes common to most operating systems. Some operating systems do not make use of some of these attributes and where that is the case, these bits can simply be ignored.

Table 14. FILE Attributes

Name	Description	Value
FILE_READ_ONLY_BIT	This bit is set if the file is marked as read only.	BIT8
FILE_HIDDEN_BIT	This bit is set if the file is hidden from the user.	BIT9
FILE_SYSTEM_BIT	This bit is set if the file is a system file.	BIT10
FILE_MODIFIED_BIT	This bit is set if the file has been modified. This is also referred to as an “archive” flag.	BIT11
FILE_IN_USE_BIT	This bit set if the file was in use at the time it was backed up.	BIT16
FILE_NAME_IN_STREAM_BIT	This bit set if the file name is stored in a stream associated with this DBLK.	BIT17
FILE_CORRUPT_BIT	This bit set if the data associated with the file could not be read.	BIT18
	Reserved (set to zero)	BIT0 – BIT7 BIT12 - BIT15 BIT19 - BIT23
	Vendor Specific	BIT24 - BIT31

Last Modification Date {5 bytes}

The *Last Modification Date* field is five bytes in length and contains the date and time that the file was last modified. This field uses an MTF_DATE_TIME low level structure.

Creation Date {5 bytes}

The *Creation Date* field is another five byte field using the MTF_DATE_TIME low level structure. This field contains the date and time that the file was first created.

Backup Date {5 bytes}

The *Backup Date* field is another five byte MTF_DATE_TIME field containing the date and time that the file was last backed up. A file is considered “backed up” if it is copied to removable storage media as part of a data protection operation.

Last Access {5 bytes}

The *Last Access* field also uses the five byte MTF_DATE_TIME low level structure to describe the date and time that the file was last accessed. Note: If possible, backup programs should not affect this field.

Directory ID {4 bytes}

The *Directory ID* is a four byte field containing the ID of the directory that the file resides in. This ID should be the same as that which was set in the *Directory ID* field of the last processed MTF_DIRB DBLK. This field is used for error handling and recovery. Refer to the MTF_DIRB DBLK description for information on how Directory ID’s are generated.

File ID {4 bytes}

The *File ID* field is a four byte field identifying this file. File IDs start at one (0001h) with the first file belonging to a Data Set and are incremented by one for each file processed. This field is used for error handling and recovery.

File Name {4 bytes}

The *File Name* field uses the four byte MTF_TAPE_ADDRESS low level structure to specify the location and size of the name associated with this file. The offset within MTF_TAPE_ADDRESS points to the String Storage Area at the end of the MTF_FILE DBLK where the file name string is physically stored. The file name can contain any characters and may be of any length. Refer to the description of the MTF_TAPE_ADDRESS low level structure for details.

Since the size of the file name may result in a DBLK size that is larger than the maximum allowed, larger file names would have to be stored in a separate 'FNAM' data stream. This stream must be the first data stream following the MTF_FILE DBLK.

When restoring this file to a different operating system, the name used to create the file may have to be modified to eliminate characters that are not valid for the target operating system, or to adjust the size to the maximum file name length supported by the target operating system. File names are stored without path information.

5.2.6 Corrupt Object Descriptor Block (MTF_CFIL)

It is often the case that a DBLK has already been written when it is discovered that not all of its associated data can be read due to disk corruption, network failure, etc. When this condition occurs, the portions of the stream that could not be read are padded to maintain the correct stream size.

A **Corrupt Object Descriptor Block** (MTF_CFIL DBLK) is then written to indicate that the data associated with the previous DBLK is corrupt. The MTF_CFIL DBLK contains fields for the stream number and the byte offset in that stream where the corruption began. It is not used for any kind of media error recovery.

If needed there is only one MTF_CFIL DBLK for a specific object being written to media. Any portion of the stream which cannot be read due to the corruption is replaced on the media with zeroes.

***Note**: The exact number of bytes of object data specified in the Stream Header must still be written to media.*

The reason that the object data must be padded is because most devices do not allow positioning back to the start of the stream to rewrite the *Stream Header* with a new length. If the stream is variable in length, it is not necessary to complete more than the current segment of the stream if no more valid data can be written.

Offset	Field Name	Type	Size
0 *0h*	Common Block Header	MTF_DB_HDR	52 bytes
52 *34h*	CFIL Attributes	UINT32	4 bytes
56 *3Ch*	reserved	- - -	8 bytes
64 *40h*	Stream Offset	UINT64	8 bytes
72 *48h*	Corrupt Stream Number	UINT16	2 bytes

Structure 10. Corrupt Object Descriptor Block (MTF_CFIL)

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The *DBLK Type* field within MTF_DB_HDR will be set to 'CFIL'. The MTF_CFIL DBLK does not use an OS Specific Data Area nor a String Storage Area.

CFIL Attributes {4 bytes}

The *CFIL Attributes* field is four bytes in length organized as a 32-bit field. This field specifies the attributes of the corrupt data represented by this MTF_CFIL DBLK. Only Bits 16 - 18 are defined at this time. The table below describes the meaning of the bits.

Table 15. CFIL Attributes

Name	Description	Value
CFIL_LENGTH_CHANGE_BIT	This bit is set if the file size has changed since the file was opened for the write operation.	BIT16
CFIL_UNREADABLE_BLK_BIT	This bit is set if a hard error was encountered reading the source media (hard disk). This usually indicates that the media itself is bad (i.e. bad sector).	BIT17
CFIL_DEADLOCK_BIT	This bit is set if the file was deadlocked. (i.e. On a system supporting record and file locking, it was not possible to get a region of a file unlocked within a watchdog time interval.)	BIT18
	Reserved (set to zero)	BIT0 - BIT15 BIT19 - BIT23
	Vendor Specific	BIT24 - BIT31

Stream Offset {8 bytes}

The *Stream Offset* field uses the UINT64 low level structure to indicate the byte offset into the data stream where the corruption padding begins. If the stream data is compressed, the offset of the corruption is the offset into the data when it is decompressed.

Corrupt Stream Number {2 bytes}

The *Corrupt Stream Number* is a two byte field indicating which stream in the Data Stream Section of the previous MTF_FILE DBLK contains padding for corrupt data. Keep in mind that the Data Stream Section following a MTF_FILE DBLK can contain several data streams each with a *Stream Header*. The first stream in the Data Stream Section would be 1, the second would be 2, and so on.

5.2.7 End of Set Pad Descriptor Block (MTF_ESPB)

The **End of Set Pad Descriptor Block** (MTF_ESPB DBLK) is only used when the physical block size written by the device is larger than the Format Logical Block size specified in the MTF_TAPE DBLK. When this is the case, it is possible for a Data Set to end prior to a physical boundary. This occurs even when the data associated with the last DBLK in the Data Set has been padded to a Format Logical Block boundary. The MTF_ESPB DBLK is used in this case to pad zeroes to the next physical block boundary, where a filemark is written to media followed by an MTF_ESET DBLK, marking the end of the Data Set.

Note: It is possible to achieve the same effect by extending the last SPAD in the Data Set such that it ends on a physical block boundary if it is known that you have written the last DBLK at the time the SPAD is written.

Offset	Field Name	Type	Size
0 *0h*	Common Block Header	MTF_DB_HDR	52 bytes

Structure 11. End of Set Pad Descriptor Block (MTF_ESPB)

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The *DBLK Type* field in the MTF_DB_HDR will be set to 'ESPB'.

5.2.8 End of Data Set Descriptor Block (MTF_ESET)

The **End of Data Set Descriptor Block** (MTF_ESET DBLK) used in conjunction with a filemark, denotes the end of a Data Set. The MTF_ESET DBLK duplicates the Data Set Number and Media Write Data fields of the MTF_SSET DBLK structure for this Data Set. In addition, data streams may be present for the support of Media Based Catalogs.

Offset	Content	Type	Size
0 *0h*	Common Block Header	MTF_DB_HDR	52 bytes
52 *34h*	ESET Attributes	UINT32	4 bytes
56 *38h*	Number Of Corrupt Files	UINT32	4 bytes
60 *3Ch*	Reserved for MBC	UINT64	8 bytes
68 *44h*	Reserved for MBC	UINT64	8 bytes
76 *4Ch*	FDD Media Sequence Number	UINT16	2 bytes
78 *4Eh*	Data Set Number	UINT16	2 bytes
80 *50h*	Media Write Date	MTF_DATE_TIME	5 bytes

Structure 12. End of Data Set Descriptor Block

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The following fields of the MTF_DB_HDR structure must be set to the defined value.

- The *DBLK Type* field is set to 'ESET'.
- The *Format Logical Address* field is set to zero.
- The *Control Block ID* field is continued from the data set. All MTF_ESET DBLKs in the MBC share the same *Control Block ID*.

ESET Attributes {4 bytes}

The *ESET Attributes* field is four bytes in length organized as a 32-bit field. Only Bits 0 - 5 are defined at this time. Bits 1 - 5 are used to specify what type of backup operation was used to create the Data Set immediately preceding the MTF_ESET DBLK on the media. Possible operation types include, copy, normal backup, differential backup, incremental backup and daily backup. Only one of these five bits should be set for a given Data Set. In the descriptions that follow, the "modified" flag (describing whether a file has been created or modified) is mentioned. Another name for this is the "archive" flag. Bits 6 - 23 of this field are reserved for future use.

Table 16. ESET Attributes

Name	Description	Value
ESET Transfer Bit	This bit is set if the data management operation is a "transfer". It indicates that the files in this Data Set were removed from the source media after the operation was completed.	BIT0
ESET Copy Bit	This bit is set if the operation is a "copy". The copy method copies all selected files from the primary storage to the media. The file's "modified" flag IS NOT reset afterwards.	BIT1
ESET Normal Bit	This bit is set if the backup type is "normal". The normal backup method backs up all selected files. The file's "modified" flag IS reset afterwards.	BIT2
ESET Differential Bit	This bit is set if the backup type is "differential". The differential backup method only backs up selected files having their "modified" flag set. The file's "modified" flag IS NOT reset afterwards.	BIT3
ESET Incremental Bit	This bit is set if the backup type is "incremental". The incremental backup method only backs up selected files having their "modified" flag set. The file's "modified" flag IS reset afterwards.	BIT4
ESET Daily Bit	This bit is set if the backup type is "daily". The daily backup method only backs up selected files created or modified with today's date. The file's "modified" flag IS NOT reset afterwards.	BIT5
	Reserved (set to zero)	BIT6 - BIT23
	Vendor Specific	BIT24 - BIT31

File/Directory Detail PBA {8 bytes}

The *File/Directory Detail PBA* field uses the eight byte UINT64 structure to specify the Physical Block Address of the MBC File/Directory Detail stream. This FDD stream is associated with the Data Set marked at the end by this MTF_ESET DBLK.

FDD Media Sequence Number {2 bytes}

The *FDD Media Sequence Number* field is two bytes in length and indicates the Media Sequence Number associated with the File/Directory Detail for this Data Set.

Data Set Number {2 bytes}

The *Data Set Number* field is a two byte field containing the ID number corresponding to this Data Set. This should be the same Data Set Number found in the MTF_SSET DBLK. Data Set Numbers start at "one" (0x01) with the first Data Set on the media, and are incremented by one for each new Data Set appended to the media. Refer to the MTF_SSET DBLK for more information.

Media Write Date {5 bytes}

The *Media Write Date* field uses the five byte MTF_DATE_TIME low level structure to indicate the exact date and time that this Data Set was created.

5.2.9 End of Tape Marker Descriptor Block (MTF_EOTM)

The **End Of Tape Marker Descriptor Block** (MTF_EOTM DBLK) is used to indicate that the End Of Media (EOM) was reached while writing the media and that the Media Family continues onto another media. When the EOM is reached, the write operation may be in a wide range of conditions. Each condition must be handled in a unique way. Refer to End of Media Processing.

Offset	Content	Type	Size
0 *0h*	Common Block Header	MTF_DB_HDR	52 bytes
52 *34h*	Last ESET PBA	UINT64	8 bytes

Structure 13. End of Tape Marker Descriptor Block

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The following fields of the MTF_DB_HDR structure must be set to the defined value.

- The *DBLK Type* field is set to 'EOTM'.
- The *Format Logical Address* field is set to zero.
- The *Control Block ID* field is set to zero.

Last ESET PBA {8 bytes}

The *Last ESET PBA* field uses the eight byte UINT64 structure to specify the Physical Block Address of the last full MTF_ESET written to media.

5.2.10 Soft Filemark Descriptor Block (MTF_SFMB)

The **Soft Filemark Descriptor Block** (MTF_SFMB DBLK) is used to emulate filemarks when hardware filemark support is not available. Setting the TAPE_SOFT_FILEMARK_BIT bit in the TAPE Attributes field of the MTF_TAPE DBLK enables soft Filemarks. The Soft Filemark Block Size field of the MTF_TAPE DBLK determines the size of a MTF_SFMB DBLK. The size of the MTF_SFMB DBLK must be defined so that it starts and ends on a physical block boundary. The MTF_SFMB DBLK cannot have any associated data streams. The MTF_SFMB DBLK contains an array of physical block addresses of previous filemarks. If an entry in the array is not used, it is set to a value of zero.

Offset	Content	Type	Size
0 *0h*	Common Block Header	MTF_DB_HDR	52 bytes
52 *34h*	Number of Filemark Entries	UINT32	4 bytes
56 *38h*	Filemark Entries Used	UINT32	4 bytes
60 *3Ch*	PBA of Previous Filemarks Array	UINT32	sizeof (MTF_SFMB) - 60

Structure 14. Soft Filemark Descriptor Block

Common Block Header {52 bytes}

The *Common Block Header* field is the 52 byte MTF_DB_HDR structure at the beginning of every DBLK. The following fields of the MTF_DB_HDR structure must be set to the defined value.

- The *DBLK Type* field is set to 'SFMB'.
- The *Format Logical Address* field is set to the number of Physical Blocks from the beginning of the media.
- The *Control Block ID* field is used for error recovery. The first MTF_SFMB DBLK in a Media Family has a Control Block ID value of one. All subsequent MTF_SFMB DBLKs within the Data Set will have a Control Block ID one greater than the previous MTF_SFMB DBLK's Control Block ID.

Number of Filemark Entries {4 bytes}

The *Number of Filemark Entries* field is four bytes in size and contains the total number of filemarks in the PBA of Previous Filemark Array.

Filemark Entries Used {4 bytes}

The *Filemark Entries Used* field is four bytes in size and contains number of valid filemarks in the PBA of Previous Filemark Array.

PBA of Previous Filemarks Array {4 byte elements}

The *PBA of Previous Filemarks Array* field is an array of filemark elements. Each filemark element is a 4 byte PBA of a previous filemark. The PBA of Previous Filemarks Array is cumulative. Entries are always stored in descending order. When the number of previous filemarks exceeds the number of entries in the array, the array is filled with those entries closest to End of Data (EOD). If a entry in the array is not used, it is set to a value of zero.

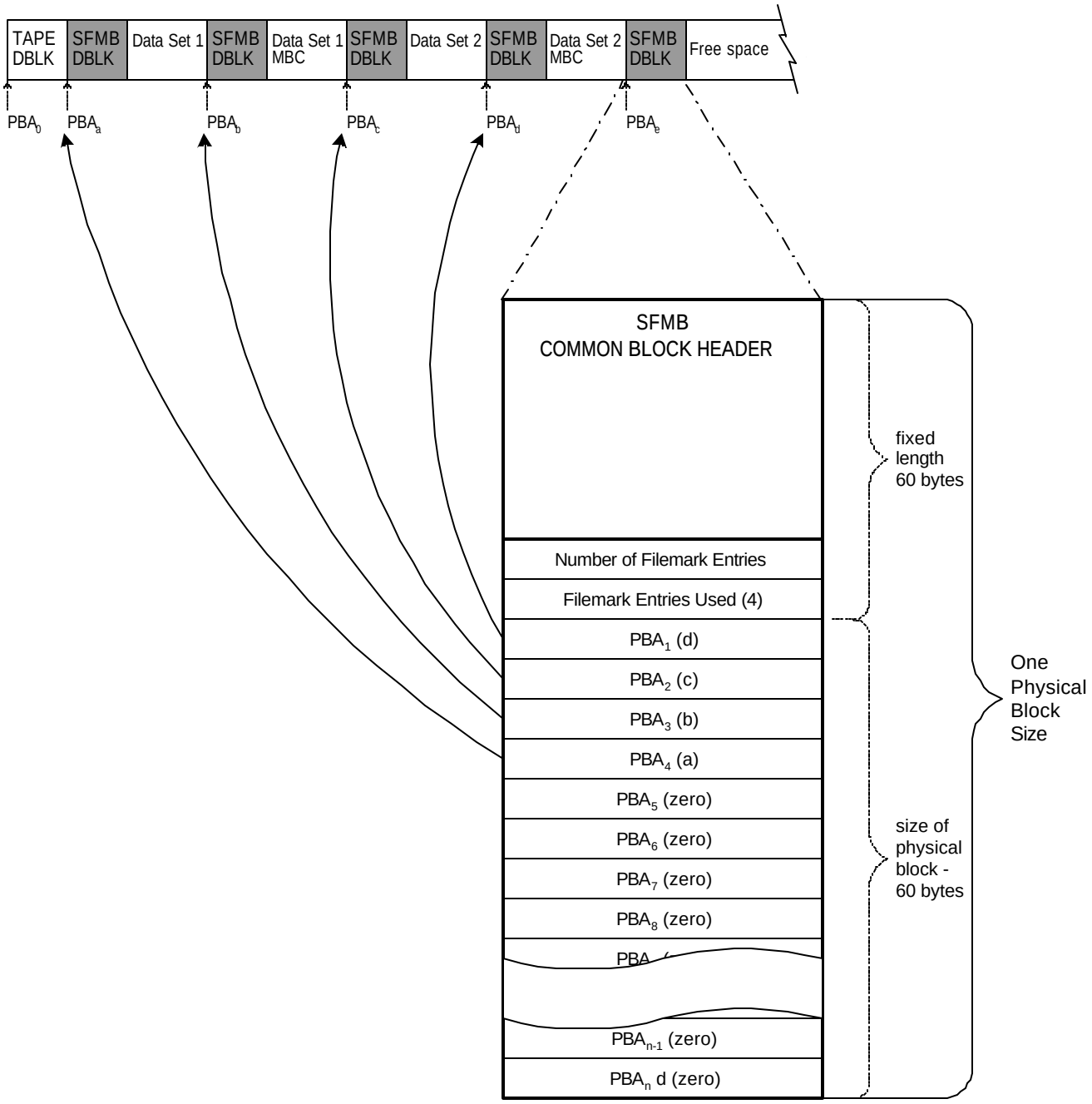


Figure 17. Soft Filemark Block Layout

Programming Note: The MTF_SFMB DBLK contains a cumulative list of filemark PBAs. To build a list of all filemarks seek to EOD and backup one physical block. Read the MTF_SFMB DBLK. If the Number of Filemark Entries field is equal to the Filemark Entries Used field, then previous MTF_SFMB DBLKs must be read to build a complete list of all filemarks.

6. Data Streams

This section provides detailed information about data streams. Data streams provided a mechanism for encapsulating different types of information using *Stream Headers*. This encapsulated information is then associated with a DBLK. One or more data streams can be associated with a DBLK. By breaking up different type of information into separate data streams, software can restore known stream types while ignoring unknown types.

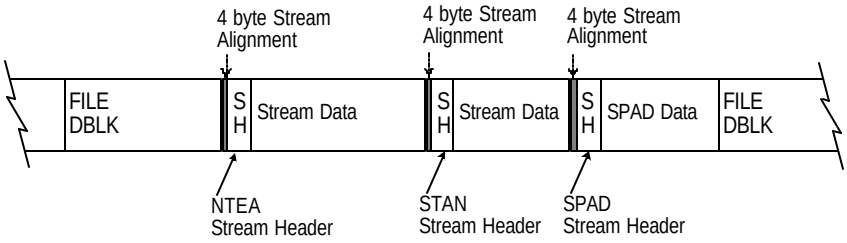


Figure 18. Data Streams

6.1 Stream Header (MTF_STREAM_HDR)

Each data stream is preceded by a ***Stream Header*** structure (MTF_STREAM_HDR). The first *Stream Header* associated with a given DBLK is located at an offset from the beginning of the DBLK. This offset is stored in the *Offset To Next Event* field of the MTF_DB_HDR portion of the DBLK. All *Stream Headers* begin on 4 byte boundaries.

If a *Stream Header* is split at EOM, it is re-written in full on the continuation media at an offset to data stored in the continuation DBLK. If EOM is crossed in the middle of the stream (by far the most common of all EOM cases), a copy of the stream's header, with an adjusted size and the continuation bit set, is written at the offset to data in the continuation DBLK. The *Stream Header* is followed by the remainder of the data.

Offset		Content	Type	Size
0	***0h***	Stream ID	UINT32	4 bytes
4	***4h***	Stream File System Attributes	UINT16	2 bytes
6	***6h***	Stream Media Format Attributes	UINT16	2 bytes
8	***8h***	Stream Length	UINT64	8 bytes
16	***10h***	Data Encryption Algorithm	UINT16	2 bytes
18	***12h***	Data Compression Algorithm	UINT16	2 bytes
20	***14h***	Checksum	UINT16	2 bytes

Structure 15. Stream Header (MTF_STREAM_HDR)

Stream ID {4 bytes}

The *Stream ID* is a four byte field that identifies the type of data stream. A four byte ASCII value as shown in the table below is used to specify the Stream ID. Additional four byte ASCII values can be added.

Stream File System Attributes {2 bytes}

The *Stream File System Attributes* field is two bytes in length and organized as sixteen bits. Only Bits 0 - 2 are defined at this time, the rest are reserved for future use. These attribute bits provide useful information about the quality of the data contained in the stream. They are defined as follows.

Table 17. Stream File System Attributes

Name	Description	Value
STREAM_MODIFIED_BY_READ	Data in stream has changed after reading, do not attempt to do a verify operation.	BIT0
STREAM_CONTAINS_SECURITY	Security information is contained in this stream.	BIT1
STREAM_IS_NON_PORTABLE	This data can only be restored to the same OS that it was saved from.	BIT2
STREAM_IS_SPARSE	The stream data is sparse (see below)	BIT3
	Reserved for future use.	BIT4 - BIT15

STREAM_IS_SPARSE

The STREAM_IS_SPARSE bit signifies that sparse data follows and is encapsulated by 'SPAR' Data Streams. The initial steam header specifies the type of sparse data in the Stream ID field (e.g., STANDARD_DATA_STREAM), has the STREAM_IS_SPARSE bit set in the Stream File System Attributes, and has a Stream Length of zero. Immediately following the initial stream header is one or more 'SPAR' Data Streams, which encapsulates the sparse data.

Stream Media Format Attributes {2 bytes}

The *Stream Media Format Attributes* field is two bytes in length and provides information about the physical characteristics of the stream as they pertain to the format. They are defined as follows.

Table 18. Stream Media Format Attributes

Name	Description	Value
STREAM_CONTINUE	This is a continuation stream.	BIT0
STREAM_VARIABLE	Data size for this stream is variable.	BIT1
STREAM_VAR_END	Last piece of the variable length data.	BIT2
STREAM_ENCRYPTED	This stream is encrypted.	BIT3
STREAM_COMPRESSED	This stream is compressed.	BIT4
STREAM_CHECKSUMED	This stream is followed by a checksum stream.	BIT5
STREAM_EMBEDDED_LENGTH	The stream length is embedded in the data	BIT6
	Reserved for future use.	BIT7 - BIT15

STREAM_EMBEDDED_LENGTH

The STREAM_EMBEDDED_LENGTH bit is obsolete and provided for backwards compatibility. This bit was used to embed the stream length when compression was active and the stream data was broken into variable length streams. This functionality is now provided in the Compression Frame Header.

Offset	Content	Type	Size
0 *0h*	Stream Length	UINT64	4 bytes
4 *4h*	Checksum	UINT16	2 bytes

Structure 16. STREAM_EMBEDDED_LENGTH

Stream Length {8 bytes}

The *Stream Length* field uses the 8 bytes UINT64 low level structure to specify the length of the current stream in bytes. The Stream Length does not include the size of the MTF_STREAM_HDR structure or any padding data used for alignment.

Data Encryption Algorithm

The *Data Encryption Algorithm* is a two byte field containing the registered ID of the encryption algorithm being used to encrypt data. This field is only important if the STREAM_ENCRYPTED bit (BIT3) of the Stream Media Format Attributes field is set.

Data Compression Algorithm

The *Data Compression Algorithm* field is a two byte field containing the registered ID of the compression algorithm being used to compress data. The STREAM_COMPRESSED bit (BIT4) of the Stream Media Format Attributes field must be set.

Checksum

The *Checksum* field contains a word-wise XOR sum of all fields from Stream ID to the Checksum field. The two byte Checksum field is not included in the checksum. This field is used to verify that a valid Stream Descriptor is being processed during read operations.

It should be noted that this checksum is not used for any file data verification. It is only used to validate the information contained in the *Stream Header* (MTF_STREAM_HDR).

6.2 Stream Data

The stream data starts immediately after the Checksum field in the *Stream Header*. This section describes the format of Stream Data for the different Stream ID types.

6.2.1 Platform Independent Stream Data

This section describes Data Streams that are Operating System independent.

Table 19. Platform Independent Stream Data Types

Name	Description	Value
STANDARD_DATA_STREAM	Standard, non-specific file data stream.	'STAN'
PATH_NAME_STREAM	Directory name in stream.	'PNAM'
FILE_NAME_STREAM	Supports extended length file names	'FNAM'
CHECKSUM_STREAM	Checksum of previous stream data.	'CSUM'

CORRUPT_STREAM	Previous stream was corrupt	'CRPT'
PAD_STREAM	Pad to next DBLK stream.	'SPAD'
SPARSE_STREAM	Sparse data.	'SPAR'
MBC_LMO_SET_MAP_STREAM	See Media Based Catalogs - Type 1	'TSMP'
MBC_LMO_FDD_STREAM	See Media Based Catalogs - Type 1	'TFDD'
MBC_SLO_SET_MAP_STREAM	See Media Based Catalogs - Type 2	'MAP2'
MBC_SLO_FDD_STREAM	See Media Based Catalogs - Type 2	'FDD2'

6.2.1.1 Standard Data Stream (STANDARD_DATA_STREAM)

The Stream ID field of the *Stream Header* is set to 'STAN' to indicate *Standard Data Stream.* The Standard Data Stream contains normal file data.

Window NT Note: When the Win32 BackupRead API is used, each data stream associated with the object being read will be preceded by a Win32 stream header. This Win32 stream header should be used to fill out the information in the MTF *Stream Header*, but **should not** be written to the media as part of the data stream.

For standard data, the dwStreamId field of the WIN32_STREAM_ID field is set to a value of BACKUP_DATA.

6.2.1.2 Directory Name In Stream (PATH_NAME_STREAM)

The Stream ID field of the *Stream Header* is set to 'PNAM' to indicate *Directory Name In Stream.* MTF limits DBLK size to the Format Logical Block Size field of the MTF_TAPE DBLK. If the directory name cannot be added to the MTF_DIRB DBLK because the new size would exceed the Format Logical Block Size limit, the directory name is placed in the first data stream associated with the MTF_DIRB DBLK. The PATH_IN_STREAM bit must be set in the DIRB Attributes field of the MTF_DIRB DBLK.

Note: When spanning, the Directory Name In Stream must be associated with the continuation MTF_DIRB DBLK.

6.2.1.3 File Name In Stream (FILE_NAME_STREAM)

The Stream ID field of the *Stream Header* is set to 'FNAM' to indicate *File Name In Stream.* MTF limits DBLK size to the Format Logical Block Size field of the MTF_TAPE DBLK. If the file name cannot be added to the MTF_FILE DBLK because the new size would exceed the Format Logical Block Size limit, the file name is placed in the first data stream associated with the MTF_FILE DBLK. The FILE_IN_STREAM bit must be set in the FILE Attributes field of the MTF_FILE DBLK.

Note: When spanning, the File Name In Stream must be associated with the continuation MTF_FILE DBLK.

6.2.1.4 Checksum Stream (CHECKSUM_STREAM)

The Stream ID field of the *Stream Header* is set to 'CSUM' to indicate *Checksum Stream.* The checksum stream is used to verify Stream Data consistency. Each data stream associated with a DBLK may have an optional checksum stream. If a data stream is going to have a checksum generated for data consistency, the Stream Media Format Attributes field of the *Stream Header* must have the STREAM_CHECKUMED bit set and the data stream that immediately follows must be the Checksum Stream (CSUM).

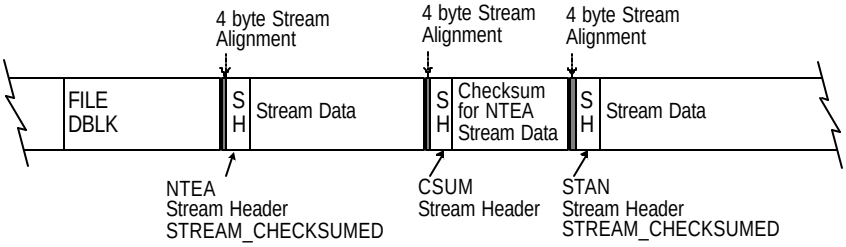


Figure 19. Checksum Stream

The checksum is a 32-bit (4 bytes) XOR sum of the linear Stream Data. Independent of how the Stream Data is segmented, the software algorithm used to generate the 32-bit checksum must guarantee consistency. For example, if the Stream Data is segmented into chunks of 1, 2, 3, or 4 bytes, the algorithm will generate the same 32-bit checksum.

6.2.1.5 Corrupt Stream (CORRUPT_STREAM)

The Stream ID field of the *Stream Header* is set to 'CRPT' to indicate the previous data stream is a *Corrupt Stream.* The Corrupt Stream is used in conjunction with the MTF_CFIL DBLK to identify which streams are corrupt. The MTF_CFIL DBLK has a limitation and can only identify a single data stream as corrupt. The Corrupt Stream has a one to one correspondence with corrupt data streams. The corrupt stream does not have any Stream Data.

6.2.1.6 Pad Stream (PAD_STREAM)

The Stream ID field of the *Stream Header* is set to 'SPAD' to indicate a *Pad Stream.* The Pad Stream is always the last data stream associated with a DBLK. The Pad Stream is used to indicate no additional data stream for the associated DBLK and brings alignment to a Format Logical Block where the next DBLK or filemark is placed. If the Pad Stream precedes a filemark, the Pad Stream must also bring alignment to a Physical Block. The Stream Data of the Pad Stream is set to NULL (binary zero) to maintain a C2 security level.

Note: Early drafts of the MTF Version 1.0 specification did not require the Pad Stream. In this case the *Offset To Next Event* field of the MTF_DB_HDR could point to a DBLK and not a *Stream Header*. In this case you should read the size of a *Stream Header* and verify the checksum. If the checksum matches make the assumption it is a *Stream Header*. If the checksum does not match continue reading up to the size of a MTF_DB_HDR and check the checksum. If the checksum matches make the assumption it is a DBLK. If the checksum does not match use error recovery to try and find the next DBLK.

6.2.1.7 Sparse Stream (SPARSE_STREAM)

The Stream ID field of the *Stream Header* is set to 'SPAR' to indicate a *Sparse Stream.* A Sparse Frame Header immediately follows the stream header and is included in the Stream Length. The Sparse Frame Header specifies the offset within the sparse file. The length of the sparse data is the Stream Length minus the size of the Sparse Frame Header.

Offset	Content	Type	Size
0 0*h*	Offset within sparse file.	UINT64	8 bytes

Structure 17. Sparse Frame Header

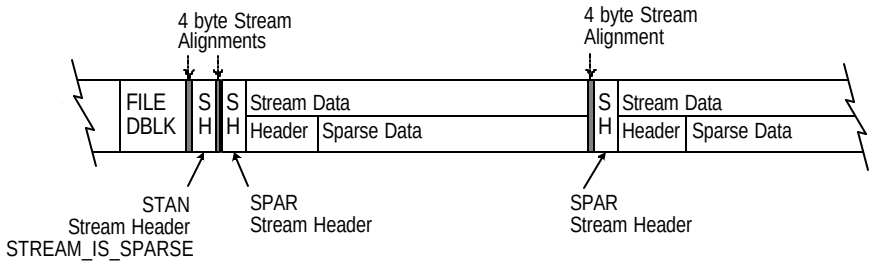


Figure 20. Windows 95 Registry Stream

6.2.2 Windows NT Stream Data

This section describes Data Streams that are specific to the Windows NT Operating System. Most NT streams are sourced using the Win32 BackupRead API. Preceding the data is a Win32 stream header (WIN32_STREAM_ID) that specifies the type of data that follows. The Win32 stream header is used to fill out the information in the MTF Stream Header, but **should not** be written as part of the Stream Data.

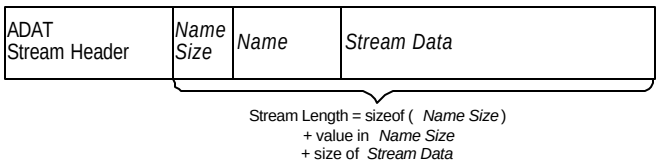
Table 20. Windows NT Stream Data Types

Name	Description	Value
STANDARD_DATA_STREAM	See definition above for Windows NT issues.	'STAN'
SPARSE_STREAM	Windows NT sparse files BACKUP_SPARSE_DATA use the platform independent SPARSE_STREAM.	'SPAR'
NTFS_ALT_STREAM	NT alternate data stream.	'ADAT'
NTFS_EA_STREAM	NT extended attribute data stream.	'NTEA'
NT_SECURITY_STREAM	NT specific security data stream.	'NACL'
NT_ENCRYPTED_STREAM	NT encrypted data stream.	'NTED'
NT_QUOTA_STREAM	NT quota data stream.	'NTQU'
NT_PROPERTY_STREAM	NT property data stream.	'NTPR'
NT_REPARSE_STREAM	NT reparse data stream.	'NTRP'
NT_OBJECT_ID_STREAM	NT object ID data stream.	'NTOI'

6.2.2.1 Windows NT Alternate Data (NTFS_ALT_STREAM)

The Stream ID field of the *Stream Header* is set to 'ADAT' to indicate *Windows NT Alternate Data*. For Windows NT Alternate Data, the dwStreamId field of the WIN32_STREAM_ID field is set to a value of BACKUP_ALTERNATE_DATA.

Windows NT Alternate Data streams require some special processing in MTF. The MTF *Stream Header* for these streams is written as usual, with the stream type being 'ADAT'. However, a 4 byte stream name size field and the stream name should precede the actual data in the MTF data stream, with the 4 bytes for the size field and the size it contains added to the total size of the data stored in the MTF *Stream Header*. The Win32 stream header for these streams contains a stream name size field, and is followed by the stream name. The string type for this name is always UNICODE, and the size is the size in bytes, **not** including a null terminator.



6.2.2.2 Windows NT Extended Attribute Data (NTFS_EA_STREAM)

The Stream ID field of the *Stream Header* is set to 'NTEA' to indicate *Windows NT Extended Attribute Data*. For Windows NT Extended Attribute Data, the dwStreamId field of the WIN32_STREAM_ID field is set to a value of BACKUP_EA_DATA.

6.2.2.3 Windows NT Security Data (NT_SECURITY_STREAM)

The Stream ID field of the *Stream Header* is set to 'NACL' to indicate *Windows NT Security Data*. For Windows NT Security Data, the dwStreamId field of the WIN32_STREAM_ID field is set to a value of BACKUP_SECURITY_DATA.

Note: The Stream File System Attributes field of the *Stream Header* must have the STREAM_CONTAINS_SECURITY bit set.

6.2.2.4 Windows NT Encrypted Data (NT_ENCRYPTED_STREAM)

The Stream ID field of the *Stream Header* is set to 'NTED' to indicate *Windows NT Encrypted Data*. Data obtained through Windows NT Encryption APIs and not BackupRead.

6.2.2.5 Windows NT Quota Data (NT_QUOTA_STREAM)

The Stream ID field of the *Stream Header* is set to 'NTQU' to indicate *Windows NT Quota Data*. Data obtained through Windows NT Quota API and not BackupRead.

6.2.2.6 Windows NT Property Data (NT_PROPERTY_STREAM)

The Stream ID field of the *Stream Header* is set to 'NTPR' to indicate *Windows NT Property Data*. For Windows NT Property Data, the dwStreamId field of the WIN32_STREAM_ID field is set to a value of BACKUP_PROPERTY_DATA.

6.2.2.7 Windows NT Reparse Data (NT_REPARSE_STREAM)

The Stream ID field of the *Stream Header* is set to 'NTRP' to indicate *Windows NT Reparse Data*. For Windows NT Reparse Data, the dwStreamId field of the WIN32_STREAM_ID field is set to a value of BACKUP_REPARSE_DATA.

6.2.2.8 Windows NT Object ID Data (NT_OBJECT_ID_STREAM)

The Stream ID field of the *Stream Header* is set to 'NTOI' to indicate *Windows NT Object ID Data*. For Windows NT Object ID Data, the dwStreamId field of the WIN32_STREAM_ID field is set to a value of BACKUP_OBJECT_ID.

6.2.3 Windows 95 Stream Data

This section describes Data Streams that are specific to the Windows 95 Operating System.

6.2.3.1 Windows 95 Registry Stream (WIN95_REGISTRY_STREAM)

The Stream ID field of the *Stream Header* is set to 'GERC' to indicate a *Windows 95 Registry Stream.* The Windows 95 Registry Streams are associated with the root directory (MTF_DIRB DBLK) of the volume in which the Windows 95 Operating System resides. The HKEY_LOCAL_MACHINE and HKEY_USERS are stored in separate Windows 95 Registry Streams. The Stream Length field of the *Stream Header* is set to the size of the GERC_HEADER plus the length of the HKEY_XXX data.

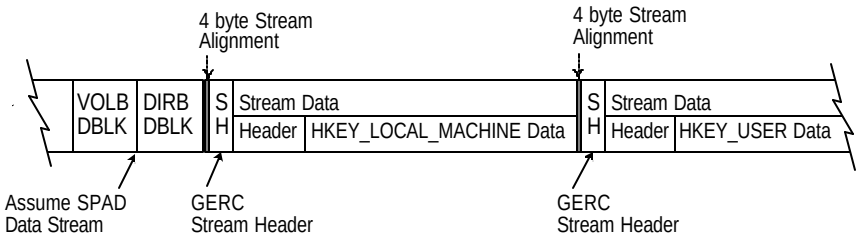


Figure 21. Windows 95 Registry Stream

The GERC_HEADER is a 260 byte header that contains the key root handle and key name.

Offset	Content	Type	Size
0 *0h*	Key ID	UINT32	4 bytes
4 *4h*	Key Name	UINT8 [256]	256 bytes

Structure 18. Windows 95 Registry Stream

Key ID {4 bytes}

The *Key ID* is a four byte field that identifies the type of registry data in the Stream Data. The Key ID field must be set to HKEY_LOCAL_MACHINE or HKEY_USERS.

Key Name {4 bytes}

The *Key Name* is a null-terminated string containing the name of the key in the Stream Data. If the Key Name string is zero length then they key being backed up is the root key itself.

6.2.4 NetWare Stream Data

This section describes Data Streams that are specific to the Novell NetWare Operating System.

Table 21. NetWare Stream Data Types

Name	Description	Value
NETWARE_386_TRUSTEE_STREAM	NetWare trustee information	'N386'
NETWARE_BINDERY_STREAM	NetWare bindery	'NBND'
NETWARE_SMS_DATA_STREAM	NetWare SMS data format	'SMSD'

6.2.4.1 NetWare Trustee Information (NETWARE_386_TRUSTEE_STREAM)

The Stream ID field of the *Stream Header* is set to 'N386' to indicate *NetWare 286 or NetWare 386 Trustee Information*. The stream is a concatenation of the trustees for the file or directory in the format of a repeating sequence of a DWORD representing the trustees ID and a WORD representing the rights mask, whether maximum (NetWare 2.x) or inherited (NetWare 3.x).

The TRUSTEE_INFO is a 6 byte structure that contains the trustee ID and trustee rights mask.

Offset	Content	Type	Size
0 *0h*	Trustee ID	UINT32	4 bytes
14 *Eh*	Trustee Rights Mask	UINT16	2 bytes

Structure 19. NetWare Trustee Info

6.2.4.2 NetWare Bindery (NETWARE_BINDERY_STREAM)

The Stream ID field of the *Stream Header* is set to 'NBND' to indicate *NetWare Bindery*. The NetWare Bindery stream is used for both NetWare 286 and NetWare 386. The NetWare Bindery Stream is associated with the root directory (MTF_DIRB DBLK) of the SYS volume. The Stream Data consists of one ore more BINDERY_HEADER structures and corresponding bindery source file data.

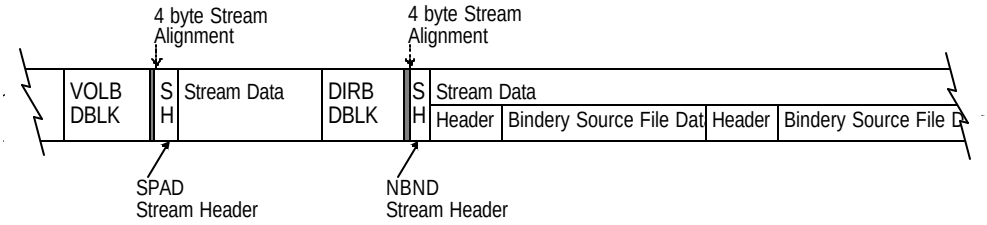


Figure 22. NetWare Bindery Stream

The BINDRY_HEADER is a 18 byte header that contains the bindery source file name and bindery source file size.

Offset	Content	Type	Size
0 *0h*	Bindery Source File Name	UINT8 [14]	14 bytes
14 *Eh*	Bindery Source File Size	UINT32	4 bytes

Structure 20. NetWare Bindery Stream

Bindery Source File Name {14 bytes}

The *Bindery Source File Name* is a null-terminated string of the bindery source file (e.g., NET$BND.SYS).

Bindery Source File Size {4 bytes}

The *Bindery Source File Size* is a four byte field containing the size of the bindery source file.

6.2.4.3 NetWare SMS Data Format (NETWARE_SMS_DATA_STREAM)

The Stream ID field of the *Stream Header* is set to 'SMSD' to indicate *NetWare SMS Data Format*. The SMS data is stored without modification.

6.2.5 OS/2 Stream Data

This section describes Data Streams that are specific to the OS/2 Operating System.

Table 22. OS/2 Stream Data Types

Name	Description	Value

HPFS_SECURITY_STREAM	HPFS security data stream.	'OACL'
HPFS_EA_STREAM	HPFS extended attribute data stream.	'O2EA'

(to be written)

6.2.6 Macintosh Stream Data

This section describes Data Streams that are specific to the Macintosh Operating System.

Table 23. Macintosh Stream Data Types

Name	Description	Value
MAC_RESOURCE_STREAM	Macintosh resource fork stream.	'MRSC'
MAC_PRIVILEGE_STREAM	Macintosh privilege stream.	'MPRV'
MAC_INFO_STREAM	Macintosh Get Info stream.	'MINF'

6.2.6.1 Macintosh Resource Stream (MAC_RESOURCE_STREAM)

The Stream ID field of the *Stream Header* is set to 'MRSC' to indicate *Macintosh Resource Stream.* The Macintosh Resource Stream contains the item's resource fork.

6.2.6.2 Macintosh Privilege Stream (MAC_PRIVILEGE_STREAM)

The Stream ID field of the *Stream Header* is set to 'MPRV' to indicate *Macintosh Privilege Stream.* The Macintosh Privilege Stream contains the privilege information for directories (foreign privilege systems, contain files as well). The first entry, "Foreign Privilege Model", contains 0 (Indicating Native Privileges) or else a number that identifies the foreign file system privilege model (only A/UX has been defined so far).

Native Privileges use the following structure:

Offset	Content	Type	Size	Apple OS Equivalent
0	Foreign Privilege Model (zero)	UINT8 [2]	2 bytes	GetVolParmsInfoBuffer . vMForeignPrivID
2	Access Rights	UINT8 [4]	4 bytes	AccessParam . ioACAccess
6	UserIDNumber	MAC_UINT32	4 bytes	AccessParam . ioACOwnerID
10	User Name	MAC_STR31	32 bytes	PBHMapID of user id number, yielding ObjParam . ioObjNamePtr
42	Group ID Number	MAC_UINT32	4 bytes	AccessParam . ioACGroupID
46	Group Name	MAC_STR31	32 bytes	PBHMapID of group id number, yielding ObjParam . ioObjNamePtr

Structure 21. Macintosh Native Privilege

Foreign Privileges use the following structure:

Offset	Content	Type	Size	Apple OS Equivalent
0	Foreign Privilege Model (non-zero)	UINT8 [2]	2 bytes	GetVolParmsInfoBuffer . vMForeignPrivID
2	Foreign Privilege Info1	UINT8 [4]	4 bytes	ForeignPrivParam . ioForeignPrivInfo1
6	Foreign Privilege Info2	UINT8 [4]	4 bytes	ForeignPrivParam . ioForeignPrivInfo2
10	Foreign Privilege Info3	UINT8 [4]	4 bytes	ForeignPrivParam . ioForeignPrivInfo3
14	Foreign Privilege Info4	UINT8 [4]	4 bytes	ForeignPrivParam . ioForeignPrivInfo4
18	Foreign Privilege Variable-Length Info	UINT8 [*n*]	*n* bytes	ForeignPrivParam . ioForeignPrivBuffer- length is ForeignPrivParam . ioForeignPrivReqCount

Structure 22. Macintosh Foreign Privilege

It is allowed (but not required) for a backup application to include both the Native and Foreign versions of this information for a given directory by including two MAC_PRIVILEGE_STREAMS after its MTF_DIR DBLK. Apple does not currently support Native permissions for files.

6.2.6.3 Macintosh Info Stream (MAC_INFO_STREAM)

The Stream ID field of the *Stream Header* is set to 'MINF' to indicate *Macintosh Info Stream.* The Macintosh Info Stream contains the Get Info comments entered by a user for a given directory or file. (Comments entered for a Volume are actually implemented as comments for the volume's root directory.) The stream need not be included if there is no user comment for an item.

Offset	Content	Type	Size	Apple OS Equivalent
0	Comment Length	MAC_UINT8	1 byte	DTPBRec.ioDTBuffer [0]
1	Comment Text	UINT8 [n]	n bytes	& DTPBRec.ioDTBuffer [1]

Structure 23. Macintosh Info

6.3 Variable Length Streams

Variable Length Streams are used to segment stream data. Each segment of the stream data is encapsulated by a *Stream Header*. Every *Stream Header* used to make up a variable length stream has the STREAM_VARIABLE bit set in the Stream Media Format Attributes field. The last *Stream Header* has the STREAM_VAR_END bit set in the Stream Media Format Attributes field.

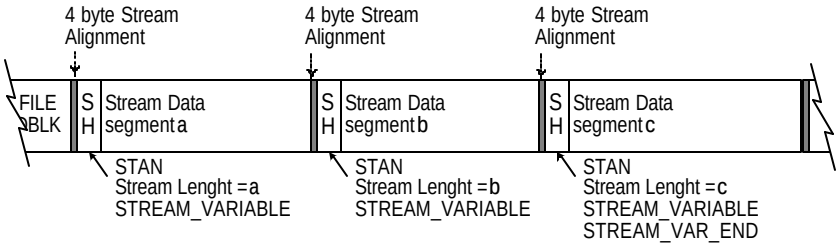


Figure 23. Variable Length Streams

6.4 Data Compression

MTF supports the compression of stream data with the exception of the Pad, Set Map, and File/Directory detail streams. When compression is enabled, all DBLKs within a backup set have the MTF_COMPRESSION bit set in the *Block Attributes* field of the MTF_DB_HDR and the *Software Compression Algorithm* field of the MTF_SSET DBLK is set to the appropriate software compression algorithm ID. Only one software compression algorithm can be used per backup set.

Note: The MTF_COMPRESSION bit and Software Compression Algorithm field is set even if no streams are compressed.

To compress stream data, the *Stream Header* must have the STREAM_COMPRESSED bit of the Stream Media Format field set and the Data Compression Algorithm field set to the same ID stored in the *Software Compression Algorithm* field of the MTF_DB_HDR. Once a *Stream Header* is set to indicate compression is active, all stream data must be encapsulated by Compression Frame Headers. Due to the nature of software compression, the streams will most likely be variable length.

When a stream is to be written with both data compression and data encryption, the data would be compressed first and then encrypted. Therefore, when reading a stream that is compressed and encrypted, the data is decrypted first and then decompressed.

6.4.1 Compression Frame Header (MTF_CMP_HDR)

The **Compression Frame Header** is used to encapsulate compressed stream data and provides all information necessary for decompression. It must also provide the total number of uncompressed data bytes from the current compression frame header through the end of the stream if available.

Offset	Field Name	Type	Size
0 *00h*	Compression Header ID	UINT16	2 bytes
2 *02h*	Stream Media Format Attributes	UINT16	2 bytes
4 *04h*	Remaining Stream Size	UINT64	8 bytes
12 *0Ch*	Uncompressed Size	UINT32	4 bytes
16 *10h*	Compressed Size	UINT32	4 bytes
20 *15h*	Sequence Number	UINT8	1 byte
21 *16h*	reserved	---	1 byte
22 *17h*	Checksum	UINT16	2 bytes

Structure 24. Compression Frame Header (MTF_CMP_HDR)

Compression Header ID {2 bytes}

The *Compression Header ID* field identifies this as the start of a compression frame header. The compression header ID field contains a two character ASCII signature 'FH' (0x4846).

Stream Media Format Attributes

The *Stream Media Format Attributes* field contains the original stream media format attributes from the *Stream Header* prior to compression. After decompression, the STREAM_VARIABLE bit in the stream media format attributes can be used to mimic the original stream state.

Remaining Stream Size

The *Remaining Stream Size* field contains the uncompressed stream length in the first compression frame header. In subsequent compression frame headers this field is computed by taking the remaining stream size field from the previous compression frame header and subtracting the uncompressed size from the previous compression frame header. If the total uncompressed stream length is unavailable, this field is set to zero.

Uncompressed Size

The *Uncompressed Size* field contains the total number of uncompressed bytes encapsulated by this compression frame header.

Compressed Size

The *Compressed Size* field contains the total number of compressed bytes encapsulated by this compression frame header. Sometimes the size of compressed data is grater than that of the uncompressed data. In this case, the uncompressed data is encapsulated by the compression frame header and not the compressed data and the compressed size is set to equal the uncompressed size.

Sequence Number

The *Sequence Number* field starts with a value of 1 for the first compression frame header and is incremented in each subsequent compression frame header. Because the Sequence Number field is 1 byte in size, the sequence number will wrap every 256 frames.

Checksum

The *Checksum* field contains a word-wise XOR sum of all fields from Compression Header ID to the Checksum field. The two byte Checksum field is not included in the checksum. This field is used to verify that a valid Compression Frame Header is being processed during read operations.

6.5 Data Encryption

MTF supports the encryption of stream data with the exception of the Pad, Set Map, and File/Directory detail streams. When compression is enabled, all DBLKs within a backup set have the MTF_ENCRYPTION bit set in the *Block Attributes* field of the MTF_DB_HDR. Only one software encryption algorithm can be used per backup set.

To encrypt stream data, the *Stream Header* must have the STREAM_ENCRYPTED bit of the Stream Media Format field set and the Data Encryption Algorithm field set. Once a *Stream Header* is set to indicate encryption is active, all stream data must be encapsulated by Encryption Frame Headers.

When a stream is to be written with both data compression and data encryption, the data would be compressed first and then encrypted. Therefore, when reading a stream that is compressed and encrypted, the data is decrypted first and then decompressed.

Note: When stream data is both compressed and encrypted, the Compression Frame Headers are encrypted as if they were stream data.

6.5.1 Encryption Frame Header (MTF_ENC_HDR)

The **Encryption Frame Header** is used to encapsulate encrypted stream data and provides all information necessary for decryption. It must also provide the total number of unencrypted data bytes from the current encryption frame header through the end of the stream if available.

Offset	Field Name	Type	Size
0 *00h*	Encryption Header ID	UINT16	2 bytes
2 *02h*	Stream Media Format Attributes	UINT16	2 bytes
4 *04h*	Remaining Stream Size	UINT64	8 bytes
12 *0Ch*	Unencrypted Size	UINT32	4 bytes
16 *10h*	Encrypted Size	UINT32	4 bytes
20 *15h*	Sequence Number	UINT8	1 byte
21 *16h*	reserved	---	1 byte
22 *17h*	Checksum	UINT16	2 bytes

Structure 25. Encryption Frame Header (MTF_ENC_HDR)

Encryption Header ID {2 bytes}

The *Encryption Header ID* field identifies this as the start of a encryption frame header. The encryption header ID field contains a two character ASCII signature ‘EH’ (0x4845).

Stream Media Format Attributes

The *Stream Media Format Attributes* field contains the original stream media format attributes from the *Stream Header* prior to encryption. After decryption, the STREAM_VARIABLE bit in the stream media format attributes can be used to mimic the original stream state.

Remaining Stream Size

The *Remaining Stream Size* field contains the unencrypted stream length in the first encryption frame header. In subsequent encryption frame headers this field is computed by taking the remaining stream size field from the previous encryption frame header and subtracting the unencrypted size from the previous encryption frame header. If the total unencrypted stream length is unavailable, this field is set to zero.

Unencrypted Size

The *Unencrypted Size* field contains the total number of unencrypted bytes encapsulated by this encryption frame header.

Encrypted Size

The *Encrypted Size* field contains the total number of encrypted bytes encapsulated by this encryption frame header.

Sequence Number

The *Sequence Number* field starts with a value of 1 for the first encryption frame header and is incremented in each subsequent encryption frame header. Because the Sequence Number field is 1 byte in size, the sequence number will wrap every 256 frames.

Checksum

The *Checksum* field contains a word-wise XOR sum of all fields from Encryption Header ID to the Checksum field. The two byte Checksum field is not included in the checksum. This field is used to verify that a valid Compression Frame Header is being processed during read operations.

7. Media Based Catalog

This section provides detailed information about Media Based Catalogs (MBC). A Media Based Catalog is comprised of two parts. The first is a Set Map which contains cumulative information about each backup set in a Media Family. The second is a File/Directory Detail which contains information specific to a backup set. Both the Set Map and File/Directory Detail are stored as Data Streams associated with the MTF_ESET DBLK.

Two different implementations of the Media Based Catalog are defined in MTF Version 1.00a. The first is "Type 1" and the second is "Type 2". Either catalog type may be used but must be consistent within a Media Family. The type of MBC being used must be defined in the *Media Based Catalog Type* field of the MTF_TAPE DBLK. The version of the particular catalog type can be defined in the *MBC Version* field of the MTF_SSET DBLK.

7.1 Control Bits

Control bits within the MTF_DB_HDR attribute field of certain DBLKs are used to determine whether an attempt will be made to write the FDD and Set Map for a given Data Set and Media Family. The table below describes the bits that are used and their meanings. Please refer to the description of the MTF_DB_HDR for more information on the use of the attribute field.

Table 24. Media Based Catalog Control Bits

Bit Name	Description	Value
MTF_SET_MAP_EXISTS	Set in the MTF_TAPE DBLK to indicate that Set Map streams must be written after each Data Set in the Media Family.	BIT16
MTF_FDD_ALLOWED	Set in the MTF_TAPE DBLK to indicate that FDD streams may be written after each Data Set in the Media Family. This bit being set does not require that the FDD be written for each Data Set.	BIT17
MTF_FDD_EXISTS	Set in the MTF_SSET DBLK to indicate that an FDD stream will be written for that Data Set. If the FDD is not written the MTF_FDD_ABORTED must be set in the MTF_ESET.	BIT16
MTF_NO_ESET_PBA	Set in the MTF_EOTM if no Data Set ends on this media, and therefore, no Set Map associated with MTF_ESET.	BIT16
MTF_INVALID_ESET_PBA	Set in the MTF_EOTM if the PBA of the MTF_ESET is invalid because the device doesn't support physical block addressing.	BIT17

7.2 Status Bits

Status bits within the MTF_DB_HDR attribute field of certain DBLKs are used to determine whether an attempt to write the FDD and Set Map was successful. The table below describes the bits that are used and their meanings. Please refer to the description of the MTF_DB_HDR for more information on the use of the attribute field.

Table 25. Media Based Catalog Status Bits

Bit Name	Description	Value
MTF_FDD_ABORTED	Set in the second MTF_ESET to indicate that the FDD stream was not written due to some error.	BIT16
MTF_END_OF_FAMILY	Set in the second MTF_ESET to indicate that the Set Map stream was not written due to some error. Note that this implies that no further Data Sets may be appended to this Media Family.	BIT17

7.3 Type 1 MBC

This section describes Type 1 MBC . An overview of Media Based Catalogs can be found in *Section 3 - Format Description*. Type 1 MBC includes both File/Directory Detail (FDD) and Set Map. Both are Data Streams associated with the MTF_ESET DBLK. The FDD includes entries for the MTF_VOLB, MTF_DIRB and MTF_FILE DBLKs in a given Data Set. Type 1 MBC is designed to allow entries for other DBLK types, including vendor specific types, to be included in the FDD in such a way that any application which does not recognize a give entry can easily skip it. However, only the DBLKs mentioned above are discussed here.

To create Type 1 MBC, the *Media Based Catalog Type* field of the MTF_TAPE DBLK is set to a value of 1 and the Media Catalog Version field of the MTF_SSET DBLK is set to a value of 2.

7.3.1 Physical Layout

Both the Set Map and File/Directory Detail are written as data streams associated with the MTF_ESET, and aligned on Physical Block boundaries to allow applications to seek to them directly. The FDD is written first, and the Set Map follows. It is allowed to write both an FDD and Set Map stream, or only a Set Map stream for any given Data Set, but writing only an FDD stream is not allowed. Note that the FDD can be added selectively on a per Data Set basis, but the Set Map must be maintained for each Data Set as it is appended to the Media Family.

A second MTF_ESET DBLK is written following the Set Map stream on the next Physical Block boundary, and this is followed by the second filemark which closes out the Data Set. The Physical Block Address (PBA) of the FDD and Set Map are contained in two 8 byte fields in the second MTF_ESET DBLK. Another two byte field provides the media sequence number where the FDD begins. Please refer to the MTF_ESET DBLK description for more information on these fields.

The last entry in the FDD is a special “end entry” which allows the total size of the FDD stream to be padded to insure the Set Map stream begins on a Physical Block boundary. The gap between the end of the Set Map and the second MTF_ESET is covered using a Pad Stream, as is the gap between the second MTF_ESET and the File Mark.

When a Media Family spans multiple media, the MTF_EOTM DBLK at the end of each full medium contains the PBA of the second MTF_ESET associated with the last Data Set which was completed on that medium.

The first *Reserved for MBC* field of the second MTF_ESET is used to store the physical block address of the ‘TFDD’ *Stream Header*. The second *Reserved for MBC* field of the second MTF_ESET is used to store the physical block address of the ‘TSMP’ *Stream Header*.

The following figure illustrates the physical positioning of the catalogs.

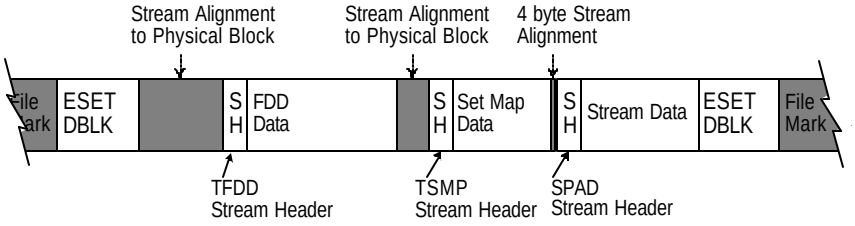


Figure 24. Physical layout of Type 1 MBC FDD and Set Map Streams

7.3.2 File/Directory Detail

The File/Directory Detail (FDD) is used to describe the volume, directory and file tree belonging to a particular Data Set. The FDD can be used to quickly determine where individual volumes, directories and files are located on the media. The FDD can be thought of as an abbreviated copy of the DBLKs in the Data Set without the data streams that follow. Only the information necessary to locate and obtain important information about individual items in the Data Set is put into the FDD stream records that describe each DBLK.

7.3.2.1 FDD Physical Layout

The FDD, like the Set Map, begins on a Physical Block boundary. The FDD is written as a stream with the Stream ID field of the *Stream Header* is set to '**TFDD**'. Please refer to the Data Stream section for further information. The MTF *Stream Header* identifies the stream as being the FDD and is followed by a series of FDD entries.

There are four types of FDD entries discussed here: **MTF_FDD_VOLB**, **MTF_FDD_DIRB**, **MTF_FDD_FILE** and **MTF_FDD_FEND**. Each entry within the FDD begins with a common header (**MTF_FDD_HDR**), and is followed by several fields of information. This is very similar to the way DBLKs use the MTF_DB_HDR at the beginning. Every FDD entry has a corresponding DBLK structure in the Data Set. The number and order of the FDD entries in the FDD match the order of the DBLKs in the Data Set which they represent, with the exception of "continuation" DBLKs written for spanning situations. The final entry in the FDD is the FEND entry.

7.3.2.2 FDD Common Header

The File/Directory Detail Common Header is a 36 byte field placed at the beginning of every FDD entry. The FDD Common Header consists of fields specifying the length of the entry, its type, the media it belongs to, and other pieces of information that are often duplicates of the corresponding DBLK that the FDD entry corresponds to.

Offset	Field Name	Type	Size
0 *0h*	LENGTH	UINT16	2 bytes
2 *2h*	TYPE	UINT8[4]	4 bytes
6 *6h*	MEDIA_SEQ_NUMBER	UINT16	2 bytes
8 *8h*	COMMON_BLOCK_ATTRIBUTES	UINT32	4 bytes
12 *0Ch*	FORMAT_LOGICAL_ADDRESS	UINT64	8 bytes
20 *14h*	DISPLAYABLE_SIZE	UINT64	8 bytes
28 *1Ch*	LINK	INT32	4 byte
32 *20h*	OS_ID	UINT8	1 byte
33 *21h*	OS_VERSION	UINT8	1 byte
34 *22h*	STRING_TYPE	UINT8	1 byte
35 *23h*	PAD	UINT8	1 byte

Structure 26. Type 1 MBC FDD Common Header (MTF_FDD_HDR)

LENGTH {2 bytes}

The *LENGTH* field indicates the size of this FDD record stream. This should equal the size of the corresponding MTF_FDD_FILE, MTF_FDD_DIRB, MTF_FDD_VOLB, or MTF_FDD_FEND record stream plus the size of any strings appended to the structure. Appended strings include names of machines, volumes, directories, and files that follow the formal field structure of the specific FDD entry.

TYPE {4 bytes}

The *TYPE* field indicates which MTF_FDD record structure this header belongs to. The *TYPE* should be "VOLB", "DIRB", "FILE", or "FEND." If the block is of type "FEND", the remaining fields in the block are undefined and should be set to zero.

MEDIA_SEQ_NUMBER {2 bytes}

The *MEDIA_SEQ_NUMBER* identifies the media in the Media Family to which this FDD belongs.

COMMON_BLOCK_ATTRIBUTES {4 bytes}

The *COMMON_BLOCK_ATTRIBUTES* field should match the corresponding field in the MTF_DB_HDR of the DBLK in the Data Set. Therefore, information about continuation, compression, end of media, variable length data streams, etc. can be found from these attribute bits.

FORMAT LOGICAL ADDRESS {8 bytes}

The *FORMAT LOGICAL ADDRESS* field matches the corresponding field in the MTF_DB_HDR of the corresponding DBLK in the Data Set. This value is used to locate the DBLK corresponding to this FDD Stream entry.

DISPLAYABLE_SIZE {8 bytes}

The *DISPLAYABLE_SIZE* field matches the corresponding field in the MTF_DB_HDR of the DBLK represented by this FDD entry. In this way, an application can quickly determine and display the size of all the files in a Data Set simply by looking at this field in the FDD entries.

LINK {4 bytes}

The *LINK* field indicates the offset of another FDD entry from the beginning of the MTF_FDD_HDR structure. *LINK* represents different offsets depending on which FDD entry it is being used in.

- File entries — The stream offset of their parent directory.
- Volume entries — The stream offset of the next volume entry, or zero for the last MTF_FDD_VOLB entry.
- Directory entries — The stream offset of the next sibling directory (i.e. next directory having same parent), or zero for the last sibling under any given parent.

OS_ID {1 byte}

The *OS_ID* field is another field that matches the corresponding field in the MTF_DB_HDR of the DBLK in the Data Set.

OS_VERSION {1 byte}

The *OS_VERSION* field also matches the corresponding field in the MTF_DB_HDR of the DBLK in the Data Set.

STRING_TYPE {1 byte}

The *STRING_TYPE* field matches the corresponding field in the MTF_DB_HDR of the DBLK represented by this FDD entry.

PAD {1 byte}

The *PAD* field is simply a one byte pad filled with zeroes to pad out to the next four byte stream alignment boundary for improved performance on RISC processors using MTF. The remaining fields of the specific FDD entry begin on this boundary.

7.3.2.3 FDD Entries

There are four record types used within the FDD. Three of them represent the volume, directory and file objects found within the Data Set that this FDD describes. Many of the data fields in these FDD entries contain duplicate copies of data found in the MTF_VOLB, MTF_DIRB and MTF_FILE DBLK fields. A fourth FDD entry called the FEND entry marks the end of the FDD. The four FDD entry types are:

Table 26. Type 1 MBC FDD Entry Types

Name	Description	Value
MTF_FDD_VOLB	FDD Volume Entry	'VOLB'

MTF_FDD_DIRB	FDD Directory Entry	'DIRB'
MTF_FDD_FILE	FDD File Entry	'FILE'
MTF_FDD_FEND	End of FDD Entry	'FEND'

7.3.2.3.1 FDD Volume Entry (MTF_FDD_VOLB)

The *FDD Volume Entry* corresponds with the VOLB DBLK it represents in the Data Set. Many of the data fields found in this structure contain copies of the data found in the VOLB DBLK fields.

Offset	Field Name	Type	Size
0 *0h*	FDD Common Header	MTF_FDD_HDR	36 bytes
36 *24h*	VOLB Attributes	UINT32	4 bytes
40 *28h*	Device Name	MTF_TAPE_ADDRESS	4 bytes
44 *2Ch*	Volume Name	MTF_TAPE_ADDRESS	4 bytes
48 *30h*	Machine Name	MTF_TAPE_ADDRESS	4 bytes
52 *34h*	OS_SPECIFIC_DATA	MTF_TAPE_ADDRESS	4 bytes
57 *38h*	Media Write Date	MTF_DATE_TIME	5 bytes

Structure 27. Type 1 MBC FDD Volume Entry (MTF_FDD_VOLB)

FDD Common Header {36 bytes}

The *FDD Common Header* field contains the 36 byte MTF_FDD_HDR structure that is found at the beginning of every FDD entry Stream. This structure was described on the preceding pages. The *TYPE* field within the MTF_FDD_HDR structure will be set to 'VOLB'.

VOLB Attributes {4 bytes}

The *VOLB Attributes* field is the same as that found in the corresponding MTF_VOLB DBLK in the Data Set. Refer to the MTF_VOLB DBLK description for information on the bits in this field.

Device Name {4 bytes}

The *Device Name* field uses the four byte MTF_TAPE_ADDRESS low level structure. This field is the same as the corresponding Device Name field in the MTF_VOLB DBLK of the Data Set, with one exception: The second two bytes used for the *Offset* field is an offset from the start of this MTF_FDD_VOLB entry to the start of the string containing the Device Name.

Volume Name {4 bytes}

The *Volume Name* field also uses the four byte MTF_TAPE_ADDRESS structure and is the same as the corresponding Volume Name field in the MTF_VOLB DBLK. The *Offset* field in the MTF_TAPE_ADDRESS low level structure is an offset from the start of this MTF_FDD_VOLB entry to the start of the string containing the Volume Name.

Machine Name {4 bytes}

The *Machine Name* field also uses the four byte MTF_TAPE_ADDRESS structure and is the same as the Machine Name field in the corresponding MTF_VOLB DBLK. The *Offset* field in the MTF_TAPE_ADDRESS structure is an offset from the start of this MTF_FDD_VOLB entry to the start of the string containing the Machine Name.

OS_SPECIFIC_DATA {4 bytes}

The *OS Specific Data* field uses the four byte MTF_TAPE_ADDRESS structure. Its contents are either zero or the same as the corresponding *OS Specific Data* field in the MTF_DB_HDR structure within the MTF_VOLB DBLK. The *Offset* field contains the offset from the start of this MTF_FDD_VOLB entry to the string containing a copy of the OS information used for the corresponding MTF_VOLB DBLK written to media.

Media Write Date {5 bytes}

The *Media Write Date* field uses the five byte MTF_DATE_TIME low level structure and is the same as the Media Write Date field in the corresponding MTF_VOLB DBLK.

7.3.2.3.2 FDD Directory Entry (MTF_FDD_DIRB)

The *FDD Directory Entry* corresponds with the MTF_DIRB DBLK it represents in the Data Set. Many of the data fields found in this structure contain copies of the data found in the MTF_DIRB DBLK fields.

Offset	Field Name	Type	Size
0 *0h*	FDD Common Header	MBC_GEN_HDR	36 bytes
36 *24h*	Last Modification Date	MTF_DATE_TIME	5 bytes
41 *29h*	Creation Date	MTF_DATE_TIME	5 bytes
46 *2Eh*	Backup Date	MTF_DATE_TIME	5 bytes
51 *33h*	Last Access Date	MTF_DATE_TIME	5 bytes
56 *38h*	DIRB Attributes	UINT32	4 bytes
60 *3Ch*	Directory Name	MTF_TAPE_ADDRESS	4 bytes
64 *40h*	OS_SPECIFIC_DATA	MTF_TAPE_ADDRESS	4 bytes

Structure 28. Type 1 MBC FDD Directory Entry (MTF_FDD_DIRB)

FDD Common Header {36 bytes}

The *FDD Common Header* field contains the 36 byte MTF_FDD_HDR structure that is found at the beginning of every FDD entry Stream. This structure was described on the preceding pages. The *TYPE* field within the MTF_FDD_HDR structure will be set to 'DIRB'.

Last Modification Date {5 bytes}

The *Last Modification Date* field uses the five byte MTF_DATE_TIME structure and contains the same data as the Last Modification Date field in the corresponding MTF_DIRB DBLK.

Creation Date {5 bytes}

The *Creation Date* field is another five byte field using the MTF_DATE_TIME low level structure. This field contains the date and time when the directory was first created. The data contained here is the same as that found in the corresponding MTF_DIRB DBLK.

Backup Date {5 bytes}

The *Backup Date* field is another five byte MTF_DATE_TIME field containing the date and time that the directory was last backed up. This is the same value as that found in the corresponding MTF_DIRB DBLK in the Data Set.

Last Access Date {5 bytes}

The *Last Access Date* field also uses the five byte MTF_DATE_TIME low level structure describing the date and time that the directory was last accessed. The data found here is a duplicate of the same field in the MTF_DIRB DBLK.

DIRB Attributes {4 bytes}

The *DIRB Attributes* field is four bytes in length organized as a 32-bit field. DIRB Attributes define characteristics of the directory represented by this MTF_DIRB DBLK. This field is the same as that found in the corresponding MTF_DIRB DBLK.

Directory Name {4 bytes}

The *Directory Name* field is four bytes in length using an MTF_TAPE_ADDRESS low level structure that specifies the location and size of the name associated with this directory. The *Offset* field used in this structure specifies the offset from the beginning of this MTF_FDD_DIRB entry to the beginning of the string containing the directory name.

OS_SPECIFIC_DATA {4 bytes}

The *OS_SPECIFIC_DATA* field uses the four byte MTF_TAPE_ADDRESS structure; its contents are either zero or a copy of the data found in the corresponding field of the MTF_DB_HDR structure within the MTF_DIRB DBLK. The *Offset* field contains the offset from the start of this MTF_FDD_DIRB entry to the string containing a copy of the OS information used for the corresponding MTF_DIRB DBLK written to media.

7.3.2.3.3 FDD File Entry (MTF_FDD_FILE)

The *FDD File Entry* corresponds with the MTF_FILE DBLK it represents in the Data Set. Many of the data fields found in this structure contain copies of the data found in the MTF_FILE DBLK fields.

Offset	Field Name	Type	Size
0 ***0h***	FDD Common Header	MTF_FDD_HDR	36 bytes
36 ***24h***	Last Modification Date	MTF_DATE_TIME	5 bytes
41 ***29h***	Creation Date	MTF_DATE_TIME	5 bytes
46 ***2Eh***	Backup Date	MTF_DATE_TIME	5 bytes
51 ***33h***	Last Access Date	MTF_DATE_TIME	5 bytes
55 ***37h***	FILE Attributes	UINT32	4 bytes
60 ***3Ch***	File Name	MTF_TAPE_ADDRESS	4 bytes
64 ***40h***	OS_SPECIFIC_DATA	MTF_TAPE_ADDRESS	4 bytes

Structure 29. Type 1 MBC FDD File Entry (MTF_FDD_FILE)

FDD Common Header {36 bytes}

The *FDD Common Header* field contains the 36 byte MTF_FDD_HDR structure that is found at the beginning of every FDD entry Stream. This structure was described on the preceding pages. The *TYPE* field within the MTF_FDD_HDR structure will be set to 'FILE'.

Last Modification Date {5 bytes}

The *Last Modification Date* field uses the five byte MTF_DATE_TIME structure and contains the same data as the Last Modification Date field in the corresponding MTF_FILE DBLK.

Creation Date {5 bytes}

The *Creation Date* field also uses the MTF_DATE_TIME low level structure containing the date and time the directory was first created. The data contained here is a duplicate of the same field in the corresponding MTF_FILE DBLK.

Backup Date {5 bytes}

The *Backup Date* field is an MTF_DATE_TIME low level structure containing the date and time that the directory was last backed up. This is the same as the data found in the corresponding MTF_FILE DBLK.

Last Access Date {5 bytes}

The *Last Access Date* field is also a duplicate of the same field in the corresponding MTF_FILE DBLK.

FILE Attributes {4 bytes}

The *FILE Attributes* field is a 32-bit field containing the same data as found in the *FILE Attributes* field of the corresponding MTF_FILE DBLK in the Data Set.

File Name {4 bytes}

The *File Name* field uses the four byte MTF_TAPE_ADDRESS low level structure that specifies the location and size of the name associated with this file. The *Offset* field in this low level structure specifies the offset from the beginning of this MTF_FDD_FILE entry to the string containing the file name.

OS_SPECIFIC_DATA {4 bytes}

The *OS_SPECIFIC_DATA* field uses the four byte MTF_TAPE_ADDRESS structure. Its contents are either zero or a copy of the data found in the corresponding field of the MTF_DB_HDR structure within the MTF_FILE DBLK. The *Offset* field contains the offset from the start of this MTF_FDD_FILE entry to the string containing a copy of the OS information used for the corresponding MTF_FILE DBLK written to media.

7.3.2.3.4 End of FDD Entry (MTF_FDD_FEND)

The *End of FDD Entry* does not corresponds with a DBLK in the Data Set. It is used to indicate the end of the FDD.

Offset	Field Name	Type	Size
0 0h	FDD Common Header	MTF_FDD_HDR	36 bytes

Structure 30. Type 1 MBC FDD End Entry (MTF_FDD_FEND)

FDD Common Header {36 bytes}

The *FDD Common Header* field contains the 36 byte MTF_FDD_HDR structure that is found at the beginning of every FDD entry Stream. This structure was described on the preceding pages. The *TYPE* field within the MTF_FDD_HDR structure will be set to 'FEND'.

The FEND entry is unique in that it does not correspond to a DBLK within the Data Set and does not have a record specific section. It is used to indicate the end of the FDD entries. The space following the FEND entry is zero padded up to the next Physical Block boundary. The Length field in the FDD Common Header specifies the offset to the next PBA. Typically, the Set Map will begin at the start of the next Physical Block boundary.

7.3.3 Set Map

The Set Map is used to list all of the Data Sets of a media or Media Family. Each successive Set Map written to a media contains information about the Data Sets previously written to media. The Set Map, like the FDD, is written as a stream and can follow the FDD or be located on an alternate partition.

The Set Map is written as a stream with the Stream ID field of the *Stream Header* is set to '**TSMP**'. Please refer to the Data Stream section for information on the *Stream Header*. The *Stream Header* identifies the stream as being the Set Map stream and is followed by three distinct parts.

1) Set Map Header

2) Set Map Entries

3) Volume Entries

7.3.3.1 Set Map Physical Layout

The Set Map begins with the *Set Map Header* which specifies the number of *Set Map Entries* which follow. Each Set Map Entry is in turn followed by a number of *Volume Entries* as specified in the Set Map Entry. There is a one-to-one correspondence between the number of Set Map Entries and Volume Entries in the Set Map, and the number of MTF_SSET and MTF_VOLB DBLKs in the Media Family. This includes continuation MTF_SSET and MTF_VOLB DBLKs written during EOM processing conditions. See Appendix J for details on End Of Media and spanning information. The order in which the Set Map Entries and Volume Entries appear in the Set Map is identical to the order in which their corresponding MTF_SSET and MTF_VOLB DBLKs are written to media.

7.3.3.2 Set Map Header (MTF_SM_HDR)

The Set Map Header is an eight byte header that contains information about the Media Family to which the Set Map belongs, the number of Set Map Entries to follow, and a pad to the next stream alignment boundary.

Offset	Field Name	Type	Size
0 ***0h***	Media Family ID	UINT32	4 bytes
4 ***4h***	Number Of Set Map Entries	UINT16	2 bytes
6 ***6h***	Pad	UINT8[2]	2 bytes

Structure 31. Type 1 MBC Set Map Header (MTF_SM_HDR)

Media Family ID {4 bytes}

The *Media Family ID* field corresponds to the same field specified in the MTF_TAPE DBLK for this media. Please refer to the MTF_TAPE DBLK description for more information on this field.

Number Of Set Map Entries {2 bytes}

The *Number Of Set Map Entries* field is two bytes in length and tells how many Set Map Entry structures are to follow this Set Map Header. One Set Map Entry is written for every Data Set written to the Media Family.

Pad {2 bytes}

The *Pad* field exists to maintain 32-bit alignment. The field should be initialized to zero.

7.3.3.3 Set Map Entry (MTF_SM_ENTRY)

The Set Map Entry corresponds with the MTF_SSET DBLK it represents in the Data Set. Many of the data fields found in this structure contain copies of the data found in the MTF_TAPE, MTF_SSET and MTF_ESET DBLK fields.

Offset	Field Name	Type	Size
0 *0h*	Length	UINT16	2 bytes
2 *2h*	Media Sequence Number	UINT16	2 bytes
4 *4h*	Common Block Attributes	UINT32	4 bytes
8 *8h*	SSET Attributes	UINT32	4 bytes
12 *Ch*	SSET PBA	UINT64	8 bytes
20 *14h*	FDD PBA	UINT64	8 bytes
28 *1Ch*	FDD Media Sequence Number	UINT16	2 bytes
30 *1Eh*	Data Set Number	UINT16	2 bytes
32 *20h*	Format Logical Address	UINT64	8 bytes
40 *28h*	Number Of Directories	UINT32	4 bytes
44 *2Ch*	Number Of Files	UINT32	4 bytes
48 *30h*	Number Of Corrupt Files	UINT32	4 bytes
52 *34h*	Data Set Displayable Size	UINT64	8 bytes
60 *3Ch*	Number Of Volumes	UINT16	2 bytes
62 *3Eh*	Password Encryption Algorithm	UINT16	2 bytes
64 *40h*	Data Set Name	MTF_TAPE_ADDRESS	4 bytes
68 *44h*	Data Set Password	MTF_TAPE_ADDRESS	4 bytes
72 *48h*	Data Set Description	MTF_TAPE_ADDRESS	4 bytes
76 *4Ch*	User Name	MTF_TAPE_ADDRESS	4 bytes
80 *50h*	Media Write Date	MTF_DATE_TIME	5 bytes
85 *55h*	Time Zone	INT8	1 byte
86 *56h*	OS_ID	UINT8	1 byte
87 *57h*	OS_VERSION	UINT8	1 byte
88 *58h*	STRING_TYPE	UINT8	1 byte
89 *59h*	MTF Minor Version	UINT8	1 byte
90 *5Ah*	Media Catalog Version	UINT8	1 byte

Structure 32. Type 1 MBC Set Map Entry (MTF_SM_ENTRY)

Length {2 bytes}

The *Length* is the size of the MTF_SM_ENTRY plus the size of any appended strings.

Media Sequence Number {2 bytes}

The *Media Sequence Number* field corresponds to the Media Sequence Number field in the MTF_TAPE DBLK to which this Data Set belongs.

Common Block Attributes {4 bytes}

The *Common Block Attributes* field has the same organization as the field of the same name in the MTF_DB_HDR structure.

SSET Attributes {4 bytes}

The *SSET Attributes* field is the same as the *SSET Attributes* field in the MTF_SSET DBLK.

SSET PBA {8 bytes}

The *SSET PBA* field corresponds to the *Physical Block Address* (PBA) field in the MTF_SSET DBLK and identifies the PBA of the MTF_SSET DBLK.

FDD PBA {8 bytes}

The *FDD PBA* field contains the same information as the File/Directory Detail PBA field of the MTF_ESET DBLK. This number specifies the Physical Block Address of the FDD associated with this Data Set.

FDD Media Sequence Number {2 bytes}

The *FDD Media Sequence Number* is a duplicate of the field of the same name in the MTF_ESET DBLK.

Data Set Number {2 bytes}

The *Data Set Number* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

Number Of Directories {4 bytes}

The *Number Of Directories* field indicates the number of directories written as part of this Data Set.

Number Of Files {4 bytes}

The *Number Of Files* field indicates the number of files written as part of this Data Set.

Number Of Corrupt Files {4 bytes}

The *Number Of Corrupt Files* field indicates the number of corrupt files written as part of this Data Set.

Data Set Displayable Size {8 bytes}

The *Data Set Displayable Size* field indicates the cumulative size of the Data Set. This should be the sum of the displayable size of every file in the Data Set.

Number Of Volumes {2 bytes}

The *Number Of Volumes* field should correspond with the number of MTF_ VOLB DBLKs in the Data Set and with the number of Volume Entries that will follow this Set Map Entry (MTF_SM_ENTRY) structure.

Password Encryption Algorithm {2 bytes}

The *Password Encryption Algorithm* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

Data Set Name {4 bytes}

The *Data Set Name* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

Data Set Password {4 bytes}

The *Data Set Password* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

Data Set Description {4 bytes}

The *Data Set Description* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

User Name {4 bytes}

The *User Name* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

Media Write Date {5 bytes}

The *Media Write Date* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

Time Zone {1 bytes}

The *Time Zone* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

OS_ID (1 byte}

The *OS_ID* field is a duplicate of the field of the same name in the MTF_DB_HDR structure of the MTF_SSET DBLK.

OS_VERSION (1 byte}

The *OS_VERSION* field is a duplicate of the field of the same name in the MTF_DB_HDR structure of the MTF_SSET DBLK.

STRING_TYPE (1 byte}

The *STRING_TYPE* field specifies the format of strings stored in the Set Map. Refer to the definition of this field in the description of the MTF_DB_HDR structure.

MTF Minor Version {1 byte}

The *MTF Minor Version* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

Media Catalog Version {1 byte}

The *Media Catalog Version* field is a duplicate of the field of the same name in the MTF_SSET DBLK.

Note: All strings associated with a Set Map Entry are appended immediately after, and pointed to by the MTF_TAPE_ADDRESS entries. The *Offset* field within the MTF_TAPE_ADDRESS structure specifies offsets from the start of this MTF_SM_ENTRY structure to the string being referred to.

7.3.3.4 Volume Entry

The Volume Entry structure in the Set Map is identical to the MTF_FDD_VOLB entry in the File/Directory Detail. Please refer to the description of the MTF_FDD_VOLB earlier in this section.

7.3.3.5 End of Media Issues

It is possible to encounter EOM while writing MBC information to media. Refer to Appendix J for detailed information on End Of Media processing and the way it is handled under different conditions.

7.4 Type 2 MBC

This section describes the Type 2 Media Based Catalog. The Type 2 Media Based Catalog includes both a Set Map and File/Directory Detail (FDD). Both of these are implemented as fixed length data streams attached to the End Of Set (MTF_ESET) DBLK.

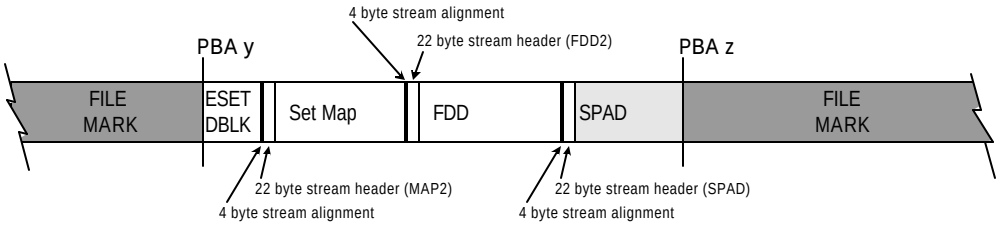


Figure 25. Physical layout of Type 2 MBC Set Map and FDD Streams

The MAP2 and FDD2 *Stream Headers* are aligned on the standard MTF stream header alignment of 4 bytes and the SPAD data stream pads to the next physical block boundary. The first *Reserved for MBC* field of the MTF_ESET is used to store the physical block address of the MTF_ESET.

To create Type 2 MBC, the Media Based Catalog Type field of the MTF_TAPE DBLK is set to a value of 2 and the Media Catalog Version field of the MTF_SSET DBLK is set to a value of 1.

7.4.1 Set Map

The Set Map is written as a stream with the Stream ID field of the *Stream Header* is set to '**MAP2**'. The *Stream Header* identifies the stream as being a Type 2 MBC Set Map and is followed by a series of DBLKs. A Type 2 Set Map is comprised of MTF_TAPE, MTF_SSET, MTF_VOLB, and MTF_ESET DBLKs. All DBLKs are packed. The *Offset To First Event* field of the MTF_DB_HDR is modified to point to the next DBLK in the data stream.

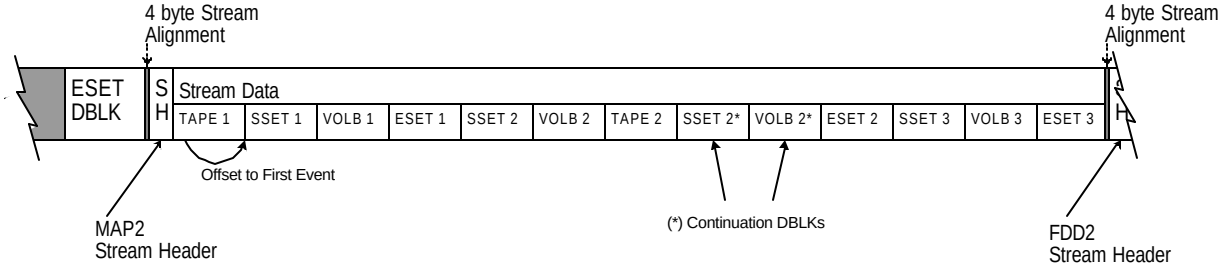


Figure 26. Type 2 MBC Set Map Example

7.4.2 File/Directory Detail

The FDD is written as a stream with the Stream ID field of the *Stream Header* is set to '**FDD2**'. The *Stream Header* identifies the stream as being a Type 2 MBC FDD and is followed by a series of DBLKs. A Type 2 FDD is comprised of MTF_VOLB, MTF_DIRB, MTF_FILE, and MTF_CFIL DBLKs. All DBLKs are packed. The *Reserved for MBC* and *Offset To First Event* fields of the Common Block Header modified. The *Reserved for MBC* is used to indicate the media number that the DBLK was written to and the *Offset To First Event* is used to point to the next DBLK in the FDD.

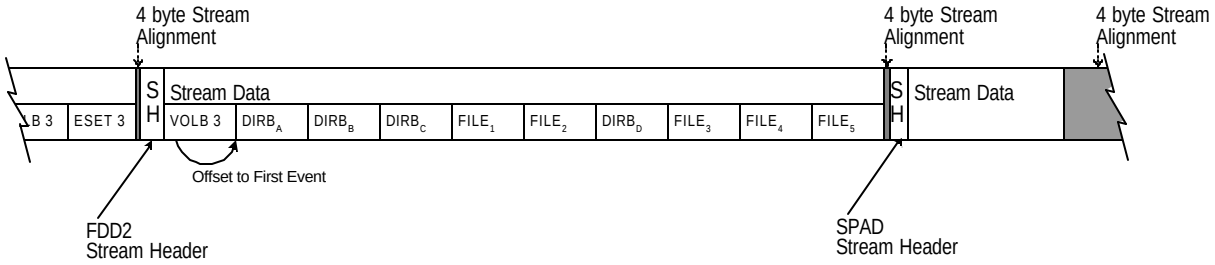


Figure 27. Type 2 MBC FDD Example

7.4.3 End of Media Issues

It is possible to encounter EOM while writing MBC information to media. Refer to the section "End of Media Processing" for detailed information on the way it is handled under different conditions.

When spanning from one media to the next, the set map is written as a data stream attached to the MTF_TAPE DBLK.

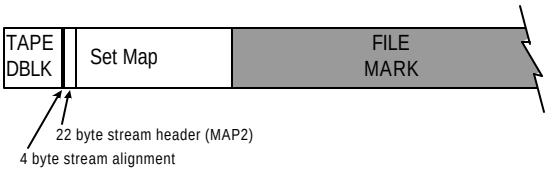
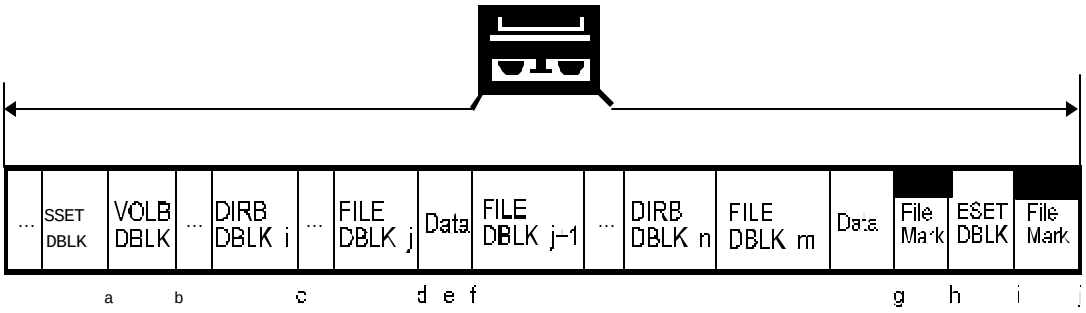


Figure 28. Type 2 MBC Spanning

8. End Of Media Processing

This section is devoted to End Of Media (EOM) processing. The following diagram is an example of a 1.0 format Data Set with marks at all unique points at which End Of Media (EOM) early warning may be detected. This is followed by diagrams and brief explanations of what is written on the original and continuation media in each case.



Before beginning the detailed explanation of how each case is handled, there are certain general concepts which need to be explained.

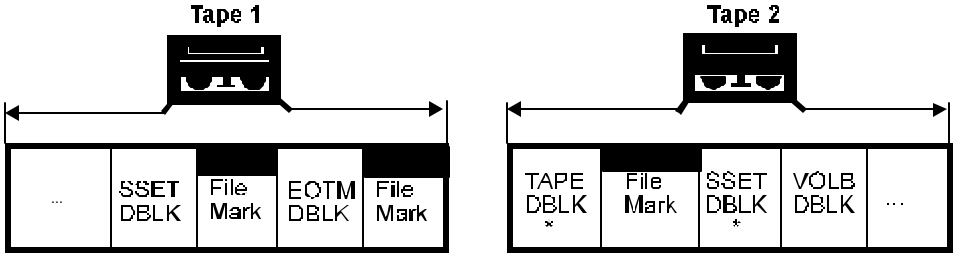
What will be referred to as "normal EOM processing" consists of writing a filemark, an End Of Tape Marker (MTF_EOTM) block and another filemark, getting a continuation tape and writing a tape header with the continuation bit set in its attribute field followed by a filemark. Any exceptions to this process will be noted in the detail for that case.

While the only block shown to have associated data is the MTF_FILE, methods for handling data associated with any block should be handled in a similar fashion. It is important to note that MTF_SSET, MTF_VOLB and MTF_DIRB blocks can be repeated on the continuation tape, with the continuation bit set in its attribute field, even when they are not the current block being processed. This is because they contain information which is necessary for reading and restoring data from the continuation tape without the need for the tape where the data management operation was started. However, if they have any associated data, it is not repeated, and the data size should be zero.

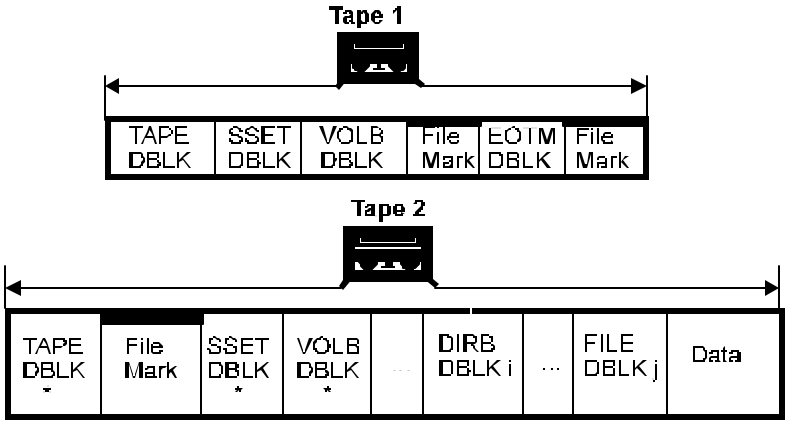
The split across EOM always occurs on Format Logical Block boundaries. For purposes of EOM processing, an image block and data is treated the same as a MTF_FILE block and data.

NOTE: In all the diagrams that follow, '*' indicates that the continuation bit (MTF_CONTINUATION) is set in the *Block Attributes* filed of the MTF_DB_HDR in the DBLK.

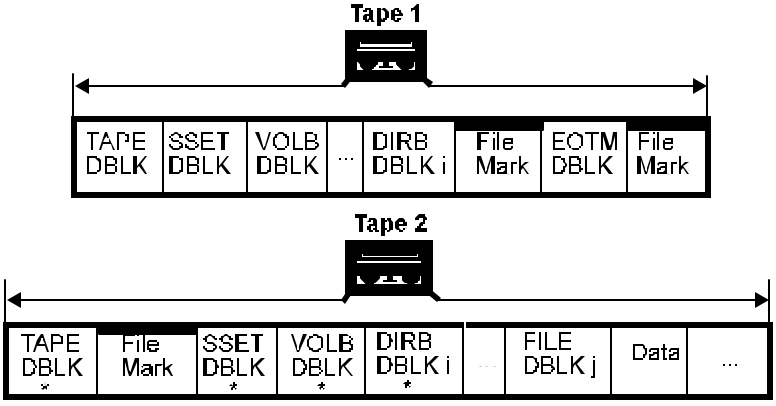
a) EOM after MTF_SSET - Process EOM normally, write the MTF_SSET again with the continuation bit set, and begin writing again from the point you left off.



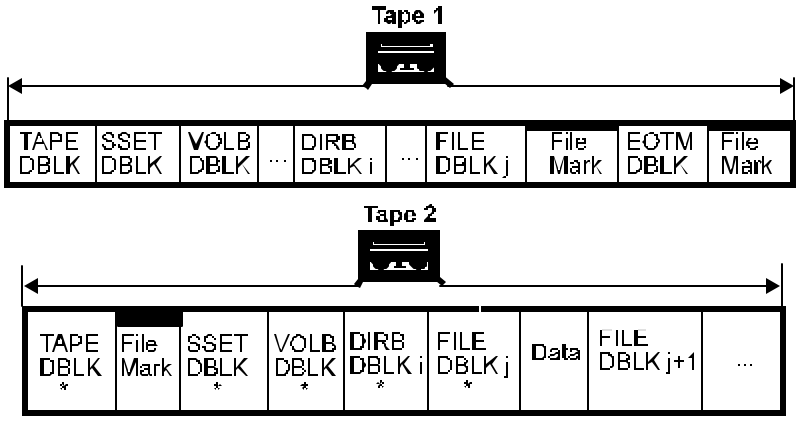
b) EOM after MTF_VOLB - Process EOM normally, write the MTF_SSET and the current MTF_VOLB again with the continuation bit set in each, and begin writing again from the point you left off.



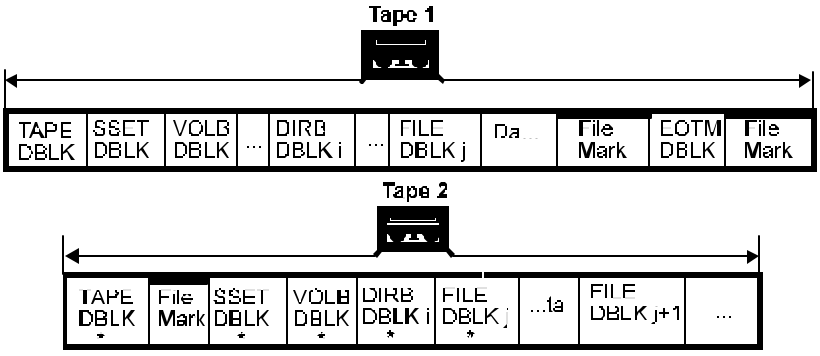
c) EOM after MTF_DIRB - Process EOM normally, write the MTF_SSET, and the current MTF_VOLB and MTF_DIRB again with the continuation bit set in all three, and begin writing again from the point you left off.



d) EOM after MTF_FILE - Process EOM normally, and write the MTF_SSET, and the current MTF_VOLB and MTF_DIRB again with the continuation bit set in all three. Write the MTF_FILE again with the continuation bit set, then write the data associated with that MTF_FILE, and continue on. Note that the data is written immediately following the MTF_FILE block, and since EOM always occurs at a Format Logical Block boundary, the chances of EOM occurring at this point are very low.

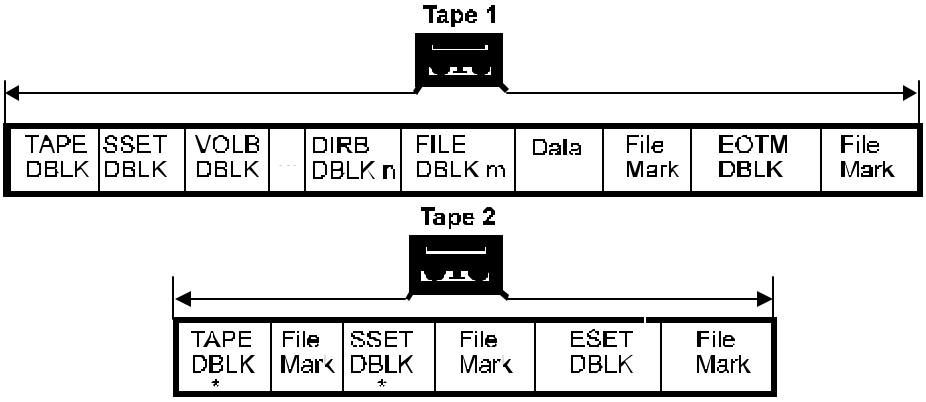


e) EOM in mid MTF_FILE data - Process the EOM in the same manner as example d. Since the data was split at a Format Logical Block boundary, and the pad at the end of the data is already calculated to align the next block on a Format Logical Block boundary, the remaining data is written beginning at the next Format Logical Block boundary, rather than flush against the end of the continuation MTF_FILE block.

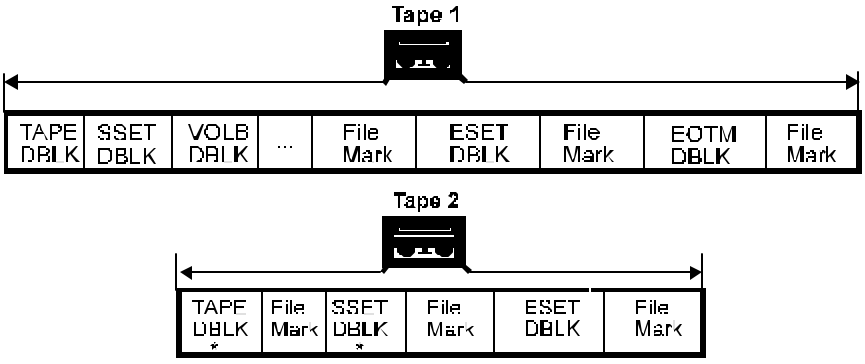


f) EOM at end of MTF_FILE data - Unlike the MTF_SSET, MTF_VOLB and MTF_DIRB, the information in the MTF_FILE block is not needed on the continuation tape if the MTF_FILE data is written completely. Therefore, writing a continuation MTF_FILE DBLK is optional, and the continuation processing is done in the same manner as example c. i.e. The continuation DBLKs are written, and then the write operation continues with the block that was due to be written when EOM occurred.

g & h) EOM at End Of Set - In this case, all set information is on tape, but the MTF_EOTM is still written as if the set continues on the next tape. Note that if the first filemark has been written, we do not write another. Only the continuation MTF_SSET needs to be written before closing out the set normally, but it must also have the bit set to indicate that the data for this set is contained fully on the previous tape.

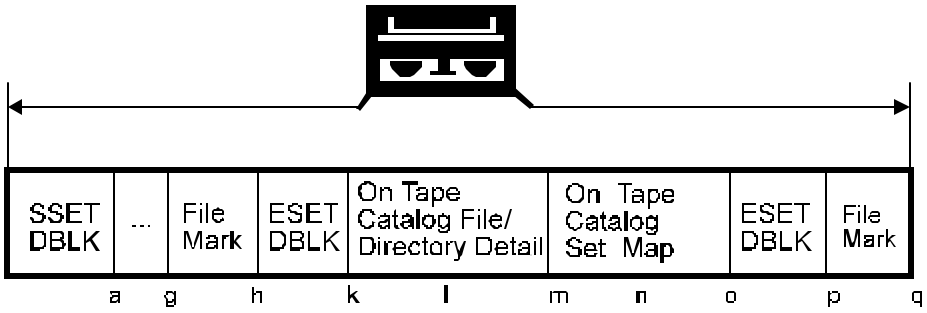


i & j) EOM between sets - In these two cases, the MTF_ESET has already been written, and the set is completed, but we do not want another set started on this tape. Therefore, we write an MTF_EOTM where the next MTF_SSET would be expected, followed by a filemark. A continuation tape is written identical to the one written for cases g & h. This is done to guarantee the existence of a unique continuation tape for beginning the next set. Note that while a MTF_TAPE DBLK alone is sufficient to mark a unique continuation tape, information such as the number of the last Data Set is necessary to append to the Media Family without requesting the previous tape.



EOM cases with Media Based Catalogs

The following diagram is an example of an MTF Version 1.00a format Data Set with Media Based Catalogs (MBC) showing marks at all unique points at which EOM early warning may be detected. This is followed by diagrams and brief explanations of what is written on the original and continuation tapes is each case. Note that the MBC lies between the two filemarks at the end of the set, and all EOM cases outside of MBC are handled in the same manner as with tapes which do not have MBC as specified above. Cases a, g, and h are shown below to relate this diagram to the non-MBC diagram above. Cases k - q are specific to MBC and detailed below.



There are some further general concepts which need to be explained before detailing the MBC cases.

In all cases, the MTF_EOTM will contain the physical block address of the second MTF_ESET of the last set which finished completely (including MBC) on the tape. Attribute bits will be defined to indicate whether the address field is invalid (not supported by drive or no MBC on tape), and to indicate if no ending MTF_ESET exists on the tape (i.e. one set spans the entire tape).

What will be referred to as "normal EOM processing" for MBC cases consists of writing a filemark, an End Of Tape Marker (MTF_EOTM) block and another filemark, getting a continuation tape and writing a tape header with the continuation bit set in its attribute field followed by a filemark, then writing the MTF_SSET with continuation bit set, another filemark, and finally the starting MTF_ESET with continuation bit set. Any exceptions to this process will be noted in the detail for that case. File/Directory Data will be referred to as FDD, and the Set Map as SM.

k) EOM after first MTF_ESET - Process EOM normally, then begin writing the FDD.

l) EOM in mid FDD - Process EOM normally, then continue writing the FDD.

m) EOM after FDD - Process EOM normally, then begin writing the SM.

n) EOM in mid Set Map - Process EOM normally. The Set Map is then rewritten from the start. The Set Map is never split between tapes!

o) EOM after Set Map - This case is handled the same way as in case n. The goal here is to make the Set Map available on the last tape in the Media Family. This makes the MBC processing a lot cleaner, and eliminates requiring the user to switch back and forth between tapes when searching for the last Set Map in a Media Family.

p & q) EOM between sets - As in cases i and j, the MTF_ESET is already written before we hit EOM, and the set is complete. So we write an MTF_EOTM where the next MTF_SSET would be expected, followed by a filemark. However, we still want a copy of the Set Map on the last tape in the Media Family. Therefore, we write the continuation tape in the same manner as case o.

Appendix A Operating System Specific Data

The *OS Specific Data* field of the MTF_DB_HDR provides a storage location for Operating System Specific Information. The *OS ID* and *OS Version* fields in the MTF_DB_HDR define the type of operating system specific data is stored in the *OS Specific Data* field. These structures are defined for their respective platforms and use native data types. All structures must be packet.

Operating System	OS ID Number	OS Version Number
NetWare	1	0
NetWare SMS	13	1
		2
Windows NT	14	0
DOS / Windows 3.X	24	0
OS/2	25	0
Windows 95	26	0
Macintosh	27	0
UNIX	28	0
To Be Assigned	33 - 127	
Vendor Specific	128 - 255	

Figure 29. OS ID and OS Version Matrix

OS ID values less than 128 may only be assigned by the MTF Review Committee. OS ID values 128-255 are reserved for vendor specific use.

NetWare (OS ID Number 1, OS Version Number 1)

The following structures are defined for *OS Specific Data* for Novell NetWare. The *OS ID* field of the MTF_DB_HDR must be set to a value of 1 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 0.

Offset		Content	Type	Size
0	*0h*	Owner ID	UINT32	4 bytes
4	*4h*	Directory Attributes	UINT32	4 bytes
8	*8h*	Maximum Space	UINT32	4 bytes
12	*Ch*	Inherited Rights	UINT16	2 bytes

Structure 33. MTF_DIRB OS Specific Data for NetWare

Offset	Content	Type	Size
0 *0h*	Owner ID	UINT32	4 bytes
4 *4h*	File Attributes	UINT32	4 bytes
8 *8h*	Last Modifer ID	UINT32	4 bytes
12 *Ch*	Archiver ID	UINT32	4 bytes
16 *10h*	Inherited Rights	UINT16	2 bytes

Structure 34. MTF_FILE OS Specific Data for NetWare

NetWare SMS (OS ID Number 13, OS Version Number 1)

The following structures are defined for *OS Specific Data* for Novell NetWare SMS. The *OS ID* field of the MTF_DB_HDR must be set to a value of 13 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 1.

Offset	Content	Type	Size
0 *0h*	Directory Attributes	UINT32	4 bytes
4 *4h*	Modified	BOOLEAN	2 bytes
6 *6h*	Creator Name Space	UINT32	4 bytes
10 *Ah*	Volume	UINT8	17 bytes

Structure 35. MTF_DIRB OS Specific Data for NetWare SMS (Version 1)

Offset	Content	Type	Size
0 *0h*	File Attributes	UINT32	4 bytes
4 *4h*	Modified	BOOLEAN	2 bytes
6 *6h*	Creator Name Space	UINT32	4 bytes
10 *Ah*	Volume	UINT8	17 bytes

Structure 36. MTF_FILE OS Specific Data for NetWare SMS (Version 1)

NetWare SME (OS ID Number 13, OS Version Number 2)

The following structures are defined for *OS Specific Data* for Novell NetWare SMS. The *OS ID* field of the MTF_DB_HDR must be set to a value of 13 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 2.

Offset	Content	Type	Size
0 *0h*	Directory Attributes	UINT32	4 bytes
4 *4h*	Creator Name Space	UINT32	4 bytes
8 *8h*	Volume	UINT8	18 bytes
26 *1Ah*	Modified	BOOLEAN	2 bytes

Structure 37. MTF_DIRB OS Specific Data for NetWare SMS (Version 2)

Offset		Content	Type	Size
0	*0h*	Directory Attributes	UINT32	4 bytes
4	*4h*	Creator Name Space	UINT32	4 bytes
8	*8h*	Volume	UINT8	18 bytes
26	*1Ah*	Modified	BOOLEAN	2 bytes

Structure 38. MTF_FILE OS Specific Data for NetWare SMS (Version 2)

Windows NT (OS ID Number 14, OS Version Number 0)

The following structures are defined for *OS Specific Data* for Windows NT. The *OS ID* field of the MTF_DB_HDR must be set to a value of 14 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 0. The Directory and File information are obtained from the WIN32_FIND_DATA structure.

Offset		Content	Type	Size
0	*0h*	Directory Attributes	UINT32	4 bytes

Structure 39. MTF_DIRB OS Specific Data for Windows NT

Offset		Content	Type	Size
0	*0h*	File Attributes	UINT32	4 bytes
4	*4h*	Short name offset	UINT16	2 bytes
6	*6h*	Short name size	UINT16	2 bytes
6	*8h*	If non-zero signifies that the file is a link to a previously written file.	BOOLEAN	2 bytes
8	*Ah*	Reserved	UINT16	2 bytes

Structure 40. MTF_FILE OS Specific Data for Windows NT

Windows NT (OS ID Number 14, OS Version Number 1)

The following structures are defined for *OS Specific Data* for Windows NT. The *OS ID* field of the MTF_DB_HDR must be set to a value of 14 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 1. The Directory and File information are obtained from the WIN32_FIND_DATA structure.

Offset		Content	Type	Size
0	*0h*	File System Flags (lpFileSystemFlags parameter from GetVolumeInformation API).	UINT32	4 bytes

4 *4h*	NT Backup Set Attributes	UINT32	4 bytes

Structure 41. MTF_VOLB OS Specific Data for Windows NT

NT Backup Set Attributes {4 bytes}

The *NT Backup Set Attributes* field is a four byte (32-bit) field specifying attributes that pertain to the NT volume. Bit 0 is defined below. Bits 1 - 23 are reserved for future use, and the most significant 8-bits (BIT24 - BIT31) are reserved for vendor specific attributes.

Table 27. TAPE Attributes

Name	Description	Value
NT_VOLB_IS_DR_CANDIDATE	If set, then the data following the VOLB should be suitable for an NT system recovery.	BIT0
	Reserved (set to zero)	BIT2 - BIT23
	Vendor Specific	BIT24 - BIT31

Offset	Content	Type	Size
0 *0h*	Directory Attributes (dwFileAttributes field of the WIN32_FIND_DATA structure)	UINT32	4 bytes
4 *4h*	Short name offset	UINT16	2 bytes
6 *6h*	Short name size	UINT16	2 bytes

Structure 42. MTF_DIRB OS Specific Data for Windows NT

Offset	Content	Type	Size
0 *0h*	File Attributes (dwFileAttributes field of the WIN32_FIND_DATA structure)	UINT32	4 bytes
4 *4h*	Short name offset	UINT16	2 bytes
6 *6h*	Short name size	UINT16	2 bytes
8 *8h*	NT File Flags (see)	UINT32	4 bytes

Structure 43. MTF_FILE OS Specific Data for Windows NT

Table 28. NT File Flags

Name	Description	Value
NT_FILE_LINK_FLAG_BIT	This bit is set if the file is a posix style hard link. If this bit is set, then the data following the DBLK should only contain one stream, this being an STRM_NTFS_LINK (“LINK”)	BIT0
	Reserved (set to zero). For backwards compatibility, these bits cannot be used.	BIT1 - BIT15
NT_FILE_POSIX_BIT	This bit is set if the file is POSIX.	BIT16
	Reserved (set to zero)	BIT17 - BIT23
	Vendor Specific	BIT24 - BIT31

DOS / Windows 3.X (OS ID Number 24, Version Number 0)

No structures are defined for DOS and Windows 3.X *OS Specific Data*. The *OS ID* field of the MTF_DB_HDR must be set to a value of 24 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 0.

OS/2 (OS ID Number 25, Version Number 0)

The following structures are defined for *OS Specific Data* for OS/2. The *OS ID* field of the MTF_DB_HDR must be set to a value of 25 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 0.

Offset	Content	Type	Size
0 *0h*	Directory Attributes	UINT32	4 bytes

Structure 49. MTF_DIRB OS Specific Data for OS/2

Offset	Content	Type	Size
0 *0h*	File Attributes	UINT32	4 bytes

Structure 50. MTF_FILE OS Specific Data for OS/2

Windows 95 (OS ID Number 26, Version Number 0)

The following structures are defined for *OS Specific Data* for Windows 95. The *OS ID* field of the MTF_DB_HDR must be set to a value of 26 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 0. The Directory and File information are obtained from the WIN32_FIND_DATA structure.

Offset	Content	Type	Size
0 *0h*	File Attributes	UINT32	4 bytes
4 *4h*	Short name offset	UINT16	2 bytes

6 *6h*	Short name size	UINT16	2 bytes

Structure 51. MTF_DIRB OS Specific Data for Windows 95

Offset	Content	Type	Size
0 *0h*	File Attributes	UINT32	4 bytes
4 *4h*	Short name offset	UINT16	2 bytes
6 *6h*	Short name size	UINT16	2 bytes

Structure 52. MTF_FILE OS Specific Data for Windows 95

Macintosh (OS ID Number 27, Version Number 0)

The following structures are defined for *OS Specific Data* for Macintosh. The *OS ID* field of the MTF_DB_HDR must be set to a value of 27 and the *OS Version* field of the MTF_DB_HDR must be set to a value of 0. A MAC_UINTXX is used to indicate 68XXX byte order.

Offset	Content	Type	Size
0 *0h*	Volume Parms Attributes	MAC_UINT32	4 bytes
4 *4h*	Volume Attributes	MAC_UINT16	2 bytes
6 *6h*	Volume Signature	MAC_UINT16	2 bytes
8 *8h*	Drive Number	MAC_UINT16	2 bytes
10 *Ah*	Driver Ref. Number	MAC_UINT16	2 bytes
12 *Ch*	File System ID	MAC_UINT16	2 bytes
14 *Eh*	Creator Data	MTF_DATE_TIME	5 bytes
19 *13h*	Modification Date	MTF_DATE_TIME	5 bytes
24 *18h*	Volume Finder Info	MAC_UINT8	32 bytes

Structure 53. MTF_VOLB OS Specific Data for Macintosh

Offset	Content	Type	Size
0 *0h*	Finder Info	MAC_UINT8	16 bytes
16 *10h*	Additional Finder Info	MAC_UINT8	16 bytes
32 *20h*	Directory ID	MAC_UINT32	4 bytes
36 *26h*	Directory Info	MAC_UINT16	2 bytes
38 *28h*	Directory X Info	MAC_UINT8	1 bytes
39 *29h*	Directory Attributes	MAC_UINT8	1 bytes

Structure 54. MTF_DIRB OS Specific Data for Macintosh

Offset		Content	Type	Size
0	*0h*	Finder Info	MAC_UINT8	16 bytes
16	*10h*	Additional Finder Info	MAC_UINT8	16 bytes
32	*20h*	Directory ID	MAC_UINT32	4 bytes
36	*24h*	File Type	MAC_UINT32	4 bytes
40	*28h*	File Creator	MAC_UINT32	4 bytes
44	*2Ch*	File Info	MAC_UINT16	2 bytes
46	*2Eh*	File X Info	MAC_UINT8	1 bytes
47	*2Fh*	File Attributes	MAC_UINT8	1 bytes

Structure 55. MTF_FILE OS Specific Data for Macintosh

UNIX (OS ID Number 28, Version Number 0)

(to be defined)

Appendix B Password Encryption Algorithm

MTF currently defines a single password encryption algorithm based on the Message Digest 5 (MD5) algorithm as described in RFC 1321. Password encryption can be done on the Media Password field of the MTF_TAPE DBLK and on the Data Set Password field of the MTF_SSET DBLK.

Table 30. Password Encryption Algorithm Table

Name	Description	Value
MTF_MD5	Message Digest 5	5

Message Digest 5

The MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. A copy of RFC 1321 can be obtained from the world wide web or via email to "mtf@smg.seagate.com".

Media Password

The Media Password field of the MTF_TAPE DBLK can be encrypted. To encrypt, the Password Encryption Algorithm field of the MTF_TAPE DBLK is set to a value of 5 (MD5 encryption algorithm). The unencrypted password is given as input to the MD5 algorithm which produces as output a 128-bit "message digest" of the password. The MD5 128-bit output is stored in the Media Password field. The Password Encryption Algorithm field is set to a value of 0 if no password encryption is used.

Note: If the Password Encryption Algorithm is unknown, no access to the media is allowed by software.

Data Set Password

The Data Set Password field of the MTF_SSET DBLK can be encrypted. To encrypt, the Password Encryption Algorithm field of the MTF_SSET DBLK is set to a value of 5 (MD5 encryption algorithm). The unencrypted password is given as input to the MD5 algorithm which produces as output a 128-bit "message digest" of the password. The MD5 128-bit output is stored in the Data Set Password field. The Password Encryption Algorithm field is set to a value of 0 if no password encryption is used.

Note: If the Password Encryption Algorithm is unknown, no access to the data set is allowed by software.

Appendix C Data Compression Algorithm

MTF currently defines a single data compression algorithm based on the Stac Technologies LZS221 compression libraries. The definition of the LZS221 compression algorithm is intended to provide cross product tape interchange of software compressed streams. It is assumed that a working knowledge of the LZS221 compression libraries is known.

Table 31. Data Compression Algorithm Table

Name	Description	Value
MTF_LZS221	Stac Technologies LZS221	0x0ABE

Common Block Header

All *Common Block Headers* in the Data Set are set to indicate possibility of compressed streams. The MTF_COMPRESSION bit is set in the *Block Attributes* field and the Software Compression Algorithm field is set to the value of 0x0ABE.

Note: Compression cannot be used on End of Set (MTF_ESET) Data Streams.

Stream Header

To indicate the stream is compressed in the ***Stream Header***, set the STREAM_COMPRESSED bit in the Stream Tape Format Attributes field and set the Data Compression Algorithm field to the value of 0x0ABE. If the compressed stream is variable length (STREAM_VARIABLE), all *Stream Headers* used to make up the variable length stream are set to indicate compression is active. Once compression is active, all stream data must be encapsulated by Compression Frame Headers.

LZS221 Buffer Sizes

The size of the buffers used by the Compress and Decompress routines are provided in the table below.

Name	Description	Size
src	Source buffer	62 * 1024
dst	Destination buffer	(62 * 1024) + 32
scratchRAM	Scratch buffer used by LZS221. This is defined in the LZS221 header file.	LZS_HISTORY_SIZE

Compress

The LZS221 compression library specifies a Compress API. Uncompressed data is passed in the src buffer and compressed data is returned in the dst buffer. The following prototype is from the LZS221-86 compression library.

```
extern void OS2_API Compress( char          **src,
                              char          **dst,
                              unsigned long  *srcCnt,
                              unsigned long  *dstCnt,
                              char          *scratchRAM);
```

Decompress

The LZS221 compression library specifies a Decompress API. Compressed data is passed in the src buffer and uncompressed data is returned in the dst buffer. The following prototype is from the LZS221-86 compression library.

```
extern int OS2_API Decompress(char          **src,
                              char          **dst,
                              unsigned long  *srcCnt,
                              unsigned long  *dstCnt,
                              char          *scratchRAM);
```

Compress and Decompress Pseudo Code

The following compress and decompress pseudo code integrates into the LZS221-86 compression library and is provided to assist in development.

```
#define STAC_CODEC_ID               0x0ABE
#define STAC_INPUT_BUFFER_SIZE      1024 * 62
#define STAC_OUTPUT_BUFFER_SIZE     (1024 * 62) + 32

m_pu8HistoryBuffer  = new UINT8 [LZS_HISTORY_SIZE];
m_pu8InputBuffer    = new UINT8 [STAC_INPUT_BUFFER_SIZE];
m_pu8OutputBuffer   = new UINT8 [STAC_OUTPUT_BUFFER_SIZE];
```

```
//_____________________________________________________________________________________________
//
// function:    StacLZS211::TryCompressing
//
// description: pass in uncompressed data and compress it.  if the compressed data is smaller
//              than the uncompressed data, the compressed data is returned.  if the compressed
//              data is larger than the uncompressed data, the uncompressed data is returned.
//
// entry:       the Uncompressed Size field of the Compression Frame Header is set to
//              psParam->u32RawDataSize.
//
//              psParam->pu8RawData             points to the buffer containing the uncompressed
//                                              data.
//              psParam->u32RawDataSize         contains the size of the uncompressed data.
//              psParam->pu8CompressedData      undefined
//              psParam->u32CompressedDataSize  undefined
//
// exit:        the Compressed Size field of the Compression Frame Header is set from
//              psParam->u32CompressedDataSize and the data pointed to by
//              psParam->pu8CompressedData is written to tape.
//_____________________________________________________________________________________________

void StacLZS211::TryCompressing (CODEC_PARAM * psParam)
{
    BOOL    fUseCompressedData;
    UINT32  u32OutputBytesUsed;
    UINT8 * pu8InputBuffer  = psParam->pu8RawData;
    UINT32  u32InputCount   = psParam->u32RawDataSize;
    UINT8 * pu8OutputBuffer = m_pu8OutputBuffer;        // local dst buffer
    UINT32  u32OutputCount  = STAC_OUTPUT_BUFFER_SIZE

    // call into the LZS221 compression libray to compress the data
    Compress ((char **)         &pu8InputBuffer,    // src
              (char **)         &pu8OutputBuffer,   // dst
              (unsigned long *) &u32InputCount,     // srcCnt (max size of STAC_INPUT_BUFFER_SIZE)
              (unsigned long *) &u32OutputCount,    // dstCnt
              (char *)          m_pu8HistoryBuffer);// scratchRAM

    if (u32OutputCount == 0)
    {
        // the compressed data is larger than the uncompressed data
        fUseCompressedData = FALSE;
    }
    else
    {
        // flush the compression buffer.  the output count must be set to
        // zero and the Compress API called again.
        u32OutputCount = 0;
        Compress ((char **)         &pu8InputBuffer,    // src
                  (char **)         &pu8OutputBuffer,   // dst
                  (unsigned long *) &u32InputCount,     // srcCnt
                  (unsigned long *) &u32OutputCount,    // dstCnt
                  (char *)          m_pu8HistoryBuffer);// scratchRAM

        // determine the number of compressed bytes in the output buffer
        u32OutputBytesUsed = pu8OutputBuffer - m_pu8OutputBuffer;

        // check to see if the compressed data is smaller than the uncompressed
        fUseCompressedData = (u32OutputBytesUsed < psParam->u32RawDataSize);
    }

    if (fUseCompressedData)
    {
        // the compressed data is smaller that the uncompressed
        psParam->pu8CompressedData      = m_pu8OutputBuffer;
        psParam->u32CompressedDataSize  = u32OutputBytesUsed;
    }
    else
    {
        // the uncompressed data is smaller that the compressed data
        psParam->pu8CompressedData      = psParam->pu8RawData;
        psParam->u32CompressedDataSize  = psParam->u32RawDataSize;
    }
}
```

```
//_____________________________________________________________________________________________
//
// function:    StacLZS211::TryDecompress
//
// description: pass in compressed data and decompress it.  the uncompressed data is returned if
//              no error condition.
//
// entry:       psParam->pu8RawData             undefined
//              psParam->u32RawDataSize         undefined
//              psParam->pu8CompressedData      points to the buffer containing all compressed
//                                              data for this frame.
//              psParam->u32CompressedDataSize  contains the size of the compressed data from the
//                                              Compressed Size field of the Compression Frame
//                                              Header.
//
// exit:        if SUCCESSFUL
//                  psParam->pu8RawData         points to a buffer containing the uncompressed
//                                              data.
//                  psParam->u32RawDataSize     contains to size of the uncompressed data in the
//                                              buffer pointed to by psParam->pu8RawData.  this
//                                              must match the Uncompressed Size field of the
//                                              Compression Frame Header.
//
//              if CODEC_ERR_COULD_NOT_DECOMPRESS
//                  psParam->pu8RawData         undefined
//                  psParam->u32RawDataSize     undefined
//_____________________________________________________________________________________________

int StacLZS211::TryDecompress (CODEC_PARAM * psParam)
{
    int     iRetVal;
    UINT8 * pu8InputBuffer  = psParam->pu8CompressedData;
    UINT32  u32InputCount   = psParam->u32CompressedDataSize;
    UINT8 * pu8OutputBuffer = m_pu8OutputBuffer;
    UINT32  u32OutputCount  = STAC_OUTPUT_BUFFER_SIZE;

    iRetVal = Decompress ((char **)         &pu8InputBuffer,    // src
                          (char **)         &pu8OutputBuffer,   // dst
                          (unsigned long *) &u32InputCount,     // srcCnt
                          (unsigned long *) &u32OutputCount,    // dstCnt
                          (char *)          m_pu8HistoryBuffer);// scratchRAM

    if (iRetVal != 0)
    {
        // if the return value from Decompress is non-zero then flush the decompressor
        u32InputCount = 0;
        iRetVal = Decompress ((char **)         &pu8InputBuffer,    // src
                              (char **)         &pu8OutputBuffer,   // dst
                              (unsigned long *) &u32InputCount,     // srcCnt
                              (unsigned long *) &u32OutputCount,    // dstCnt
                              (char *)          m_pu8HistoryBuffer);// scratchRAM
    }

    if (iRetVal == 0)
    {
        // the data was successfully decompressed
        psParam->pu8RawData     = m_pu8OutputBuffer;
        psParam->u32RawDataSize = STAC_OUTPUT_BUFFER_SIZE - u32OutputCount;

        // consistency check
        if (psParam->u32RawDataSize <= STAC_INPUT_BUFFER_SIZE)
            return SUCCESSFUL;
    }

    return CODEC_ERR_COULD_NOT_DECOMPRESS;
}
```

Appendix D Implementation Issues

Field Size and Alignment

Due to the nature of many of the 32-bit processors, all 32-bit elements are aligned on 4 byte boundaries, and all 16-bit elements are aligned on even byte boundaries. Without this requirement, the actual size of the structure would vary depending on the type of processor and the compiler. All media structures are packed to 1 byte boundaries to ensure compatibility.

Software Compression Algorithm

The Data Encryption Algorithm in the MTF_SSET DBLK has been changed to Software Compression Algorithm. MTF Version 1.00a limits a backup set to a single software compression algorithm. The addition of the Software Compression Algorithm allows software to determine if Data Sets have compatible software compression algorithms. Data Encryption has not been defined and this change should have no impact on existing products.

Block Alignment Pad

In Version 1.0 of MTF, a Block Alignment Pad is used between the end of a DBLK and the start of the next DBLK. The Block Alignment Pad can be a few bytes or hundreds of bytes depending on the length necessary to fill to the next Format Logical Block. A block alignment pad has no header, it is simply NULL data (binary zero). Block Alignment Pads are only used following DBLKs which don't have a data stream section immediately following. MTF Version 1.00a uses a *stream pad* (SPAD) in place of the Block Alignment Pad.

Offset To First Event

The *Offset To First Event* field in the ***Common Block Header*** normally points to the header of the first stream associated with that DBLK. Early drafts of the MTF 1.0 specification did not require the SPAD Data Stream. The following method is suggested for determining whether the *Offset To First Event* field points to is a *Stream Header* or DBLK:

1. Check to see if it is a known DBLK or Stream type, then use the checksum to verify the integrity of the data.
2. If step 1 fails use the checksum to determine if it is an unknown *Stream Header*.
3. If step 2 fails use the checksum to determine if it is an unknown DBLK.

Device Specific Physical Block Addressing

There are two types of positioning: absolute (physical) and logical. Not all drives support both types of positioning: the drive's feature bits indicate the type(s) supported. Also, on some drives, the media (tape) can affect whether or not positioning is supported; i.e., a drive's feature bits can change depending on whether or not there is media in the drive and/or the type of media in the drive.

Some Windows NT tape (class) drivers do a software simulation of logical tape positioning in the driver; i.e., they implement pseudo-logical tape positioning. This provides a means to write/read MTF on media and drives that intrinsically support only absolute tape positioning.

On all drives, positioning support is intended to provide a means to do a "get position" while writing to tape in order to be able to later do a "set position" to that same position and then begin reading the tape at/from that same position. This technique will always "work" and it can be done on any media and drive that supports either type of positioning; i.e., it is **always** possible to "set position" to either an absolute position or a logical position obtained by means of a "get position". It is not certain that a "set position" to a "synthetic" position (i.e., any position not directly obtained by means of a "get position") will "work". Arbitrary, "random access" positioning capability on tape is not intended; in fact, it is not supported by/on many drives.

However, it is usually possible to "set position" to a position relative to (i.e., at an offset from) a **logical** position (pseudo or real) obtained by means of a "get position". On all media and drives where **logical** positioning is supported, this relative (offset) type of "set positioning" can at least be done:

in a forward direction (positive offset),

from the logical position obtained by means of a "get position",

to/at any relative, calculated position (sum of offset and logical position obtained by means of a "get position"),

within and throughout the/any region of data that is/was consecutively and contiguously written immediately following the logical position obtained by means of a "get position" (i.e., a data "zone" produced by consecutive and contiguous "writes" and nothing else -- no other intermediate tape mark writes, "get positions", or whatever).

The foregoing is true both on media and drives that intrinsically support true (SCSI-2) logical positioning and on media and drives where (driver implemented) pseudo-logical positioning is supported.

In addition to the foregoing, on media and drives that intrinsically support true logical positioning it is possible to do both forward and reverse relative "set positions" (negative or positive offset) and relative "set positions" that cross tape mark divisions (filemarks and/or setmarks).

In the set of tape media and tape drives handled by the current set of Windows NT tape drivers (4mmdat.sys, archqic.sys, exabyte1.sys, exabyte2.sys, tandqic.sys, wangqic.sys, and qic117.sys), the media and drives that intrinsically support true (SCSI-2) logical positioning are all DAT media and drives (4mmdat.sys), the 1.35 gigabyte (9135) and 2.1 gigabyte (9210) QIC media and Archive Anaconda (model 2750 and model 2800) QIC drives (archqic.sys), and 5+ gigabyte 8mm media and SCSI-2 8mm drives (exabyte2.sys -- Exabyte 8500 series and compatible). In all other cases where logical positioning support is indicated in the drive features bits, it is pseudo-logical tape positioning support; i.e., it is done by software simulation of logical tape positioning in the driver.

Thus, although it is sometimes possible to "set position" to a position relative to (i.e., at an offset from) an **absolute** position obtained by means of a "get position", it is unnecessary to do so: pseudo-logical tape positioning is implemented in the Windows NT tape drivers where it is possible to do so. Doing so is very media/drive unique and hence requires very media/drive specific knowledge. An understanding of how this is accomplished can be acquired by studying the technical standards that govern the physical format of recorded information on the specific tape media, the tape drive technical manual(s) and the Windows NT tape driver source code. Translation between absolute position and pseudo-logical position and vice versa is accomplished by module "physlogi" in the Windows NT tape drivers. The source code for this module (physlogi.c) is included in the Windows NT DDK.

Appendix E Optical Media Framework

This appendix defines a framework for writing MTF data to an optical media This framework was defined and implemented by Seagate Software Inc. prior to the definition of the Soft Filemark Descriptor Block (MTF_SFMB DBLK).

In this framework, the optical media is viewed as a liner media. The media is divided into three parts. First is the Optical Media Header which occupies a single sector and is located at Logical Sector Address (LSA) 0x0606, the second is the Optical Data Area which starts in the next available sector after the Optical Media Header and grows towards the Optical Filemark Tables, and the third is the Optical Filemark Tables, the first of which is located at the end of the media with additional Optical Filemark Tables being added towards the Optical Data Area.

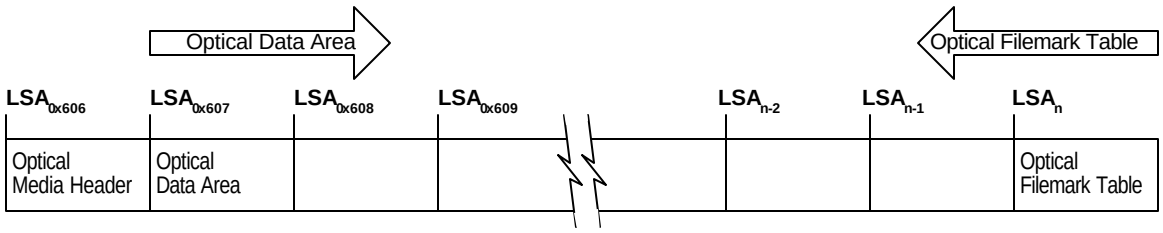


Figure 30. Optical Media Framework

Note: As the Optical Data Area and Optical Filemark Tables grow towards each other, enough space must be reserved for end of media processing.

Optical Media Header

The Optical Media Header contains signature information, starting sector of the Optical Data Area, and the starting sector of the first Optical Filemark Table.

Offset	Field Name	Type	Size
0 *0h*	Optical Signature	UINT8 [20]	21 bytes
21 *15h*	Optical Format	UINT8 [8]	9 bytes
30 *1Eh*	Volume Serial Number	UINT32	4 bytes
34 *22h*	Sector Size	UINT32	4 bytes
38 *26h*	Starting Data Sector	UINT32	4 bytes
42 *2Ah*	Starting Filemark Sector	UINT32	4 bytes

Structure 56. Optical Media Header

Optical Signature {21 bytes}

The *Optical Signature* field is a 21 byte character signature used to identify this as the Optical Media Header. The signature is set to the NULL terminated string “Arcada Software Inc.”.

Optical Format {9 bytes}

The *Optical Format* field is a 9 byte field containing a character sequence that identifies the version of the Optical Media Framework being used. This field is set to the NULL terminated string “OTEF 1.0”.

Volume Serial Number {4 bytes}

The *Volume Serial Number* is a 4 byte field that contains the volume serial number of this optical media.

Sector Size {4 bytes}

The *Sector Size* field is a 4 byte field that contains the size of a sector on this optical media.

Starting Data Sector {4 bytes}

The *Starting Data Sector* field is a 4 byte field that contains the LSA that the Optical Data Area starts.

Starting Filemark Table Sector {4 bytes}

The *Starting Filemark Table Sector* field is a 4 byte field that contains the LSA of the first Optical Filemark Table. Additional Optical Filemark Tables are added in adjacent sectors growing towards the Optical Data Area.

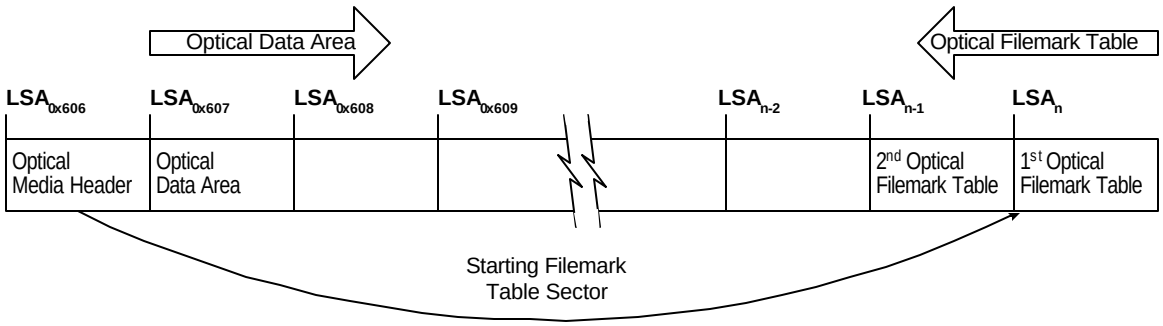


Figure 31. Multiple Optical Filemark Tables

Optical Filemark Table

The Optical Filemark Table contains the end of data sector, number of filemarks in the array, sector number of the previous filemark table, and an array of filemarks.

Offset	Field Name	Type	Size
0 *00h*	Optical Table ID	UINT32	4 bytes
4 *04h*	End Of Data Sector	UINT32	4 bytes
8 *08h*	Number Of Filemark Entries	UINT32	4 bytes
12 *0Ch*	Last Entry In Previous Filemark Table	UINT32	4 bytes
16 *10h*	Filemark Array	UINT32	sector size - 16

Structure 57. Optical Filemark Table

Optical Table ID {4 bytes}

The *Optical Table ID* field is a 4 byte character signature used to identify this as a Optical Filemark Table. This field is set to 'OTEF'. This is not a NULL terminated string.

End Of Data Sector {4 bytes}

The *End Of Data Sector* field is 4 bytes in size and contains the EOD LSA. The EOD LSA is the first free sector that can be used to grow the Optical Data Area. The End Of Data Sector field is only valid for the most recent Optical Filemark Table.

Number Of Filemark Entries {4 bytes}

The *Number Of Filemark Entries* field is 4 bytes in size and contains the number of filemark entries in the Filemark Array.

Last Entry In Previous Filemark Table {4 bytes}

The *Last Entry In Previous Filemark Table* field is 4 bytes in size and contains the LSA of the last entry in the previous Optical Filemark Table.

Filemark Array {4 bytes}

The *Filemark Array* field is an array of filemark elements. Each filemark element is a 4 byte LSA. Filemarks occupy one physical sector of undefined data. If a entry in the array is not used, it is set to a value of zero.