-OSTA.org-

Universal Disk Format® Specification

Revision 2.60

March 1, 2005

REVISION HISTORY

1.00	October 24, 1995	Original Release
1.01	November 3, 1995	DVD appendix added
1.02	August 30, 1996	Incorporates Document Change Notices DCN 2-001 through DCN 2-024
1.50	February 4, 1997	Integrated support for CD-R and CD-RW media (DCNs 2-025 thru 2-032)
2.00	April 3, 1998	Integrated support for ECMA 167 3rd Edition which included the support for named streams. (DCN 2-033 through DCN 2-064)
2.01	March 15, 2000	Incorporates DCNs 5000, 5002, 5004, 5006-5009, 5013-5015, 5018-5021, 5024-5027, 5029-5032, 5034-5042, 5044-5048, 5050
2.50	April 30, 2003	Incorporates Metadata Partition, DCNs 5049, 5061-5066, 5068-5072, 5074-5079, 5081-5082, 5086, 5089, 5090.
2.60	January 26, 2005	Incorporates Pseudo OverWrite, DCNs 5100-5121
	March 1, 2005	Approved by committee vote. Minor editorial corrections.

POINTS OF CONTACT

Optical Storage Technology Association
http://www.osta.org/

Contact information
http://www.osta.org/osta/contact.htm

Technical questions
mailto:info@osta.org

UDF Technical Editor
mailto:editor.udf@osta.org

OSTA UDF Committee email reflector
See link: “UDF Email Reflector” at the bottom of page:
http://www.osta.org/specs/index.htm

Important Notices

CONTENTS

1. Introduction

The OSTA Universal Disk Format (UDF®) specification defines a subset of the standard ECMA 167 3rd edition. The primary goal of the OSTA UDF is to maximize data interchange and minimize the cost and complexity of implementing ECMA 167.

To accomplish this task this document defines a *Domain*. A domain defines rules and restrictions on the use of ECMA 167. The domain defined in this specification is known as the "OSTA UDF Compliant" domain.

This document attempts to answer the following questions for the structures of ECMA 167 on a per operating system basis:

> *Given some ECMA 167 structure X, for each field in structure X answer the following questions for a given operating system:*
>
> > *1) When reading this field: If the operating system supports the data in this field then what should it map to in the operating system?*
> >
> > *2) When reading this field: If the operating system supports the data in this field with certain limitations then how should the field be interpreted under this operating system?*
> >
> > *3) When reading this field: If the operating system does NOT support the data in this field then how should the field be interpreted under this operating system?*
> >
> > *4) When writing this field: If the operating system supports the data for this field then what should it map from in the operating system?*
> >
> > *5) When writing this field: If the operating system does NOT support the data for this field then to what value should the field be set?*

For some structures of ECMA 167 the answers to the above questions were self-explanatory and therefore those structures are not included in this document.

In some cases additional information is provided for each structure to help clarify the standard.

This document should help make the task of implementing the ECMA 167 standard easier.

To be informed of changes to this document please fill out and return the OSTA UDF Developer Registration Form located in appendix 6.18.

1.1 Document Layout

This document presents information on the treatment of structures defined under standard ECMA 167.
This document is separated into the following 4 basic sections:

- *Basic Restrictions and Requirements* - defines the restrictions and requirements that are operating system independent.
- *System Dependent Requirements* - defines the restrictions and requirements that are operating system dependent.
- *User Interface Requirements* - defines the restrictions and requirements that are related to the user interface.
- *Informative Annex* - Additional useful information.

This document presents information on the treatment of structures defined under standard ECMA 167. The following areas are covered:

👓 Interpretation of a structure/field upon reading from media.

✍ Contents of a structure/field upon writing to media. Unless specified otherwise *writing* refers only to creating a new structure on the media. When it applies to updating an existing structure on the media it will be specifically noted as such.

The fields of each structure are listed first, followed by a description of each field with respect to the categories listed above. In certain cases, one or more fields of a structure are not described if the semantics associated with the field are obvious.

A word on terminology: in common with ECMA 167, this document will use ***shall*** to indicate a mandatory action or requirement, ***may*** to indicate an optional action or requirement, and ***should*** to indicate a preferred, but still optional action or requirement.

Also, special comments associated with fields and/or structures are prefaced by the notification: "**NOTE:**". Notes may be numbered “**NOTE 1:**”, etc.

1.2 Compliance

This document requires conformance to parts 1, 2, 3 and 4 of ECMA 167. Compliance to part 5 of ECMA 167 is not supported by this document. Part 5 may be supported in a later revision of this document.

For an implementation to claim compliance to this document the implementation shall meet all the requirements (indicated by the word *shall*) specified in this document.

The following are a few points of clarification in regards to compliance:

- *Multi-Volume support is optional*. An implementation can claim compliance and only support single volumes.
- *Multi-Partition support is optional*. An implementation can claim compliance without supporting the special multi-partition case on a single volume defined in this specification.
- *Media support*. An implementation can claim compliance and support a single media type or any combination. All implementations should be able to read any media that is physically accessible.
- *Multisession support*. Any implementation that supports reading of CD-R media shall support reading of CD-R Multisessions as defined in 6.10.3.
- *File Name Translation* - Any time an implementation has the need to transform a filename to meet operating system restrictions it shall use the algorithms specified in this document.
- *Extended Attributes* - All compliant implementations shall preserve existing extended attributes encountered on the media. Implementations shall create and maintain the extended attributes for the operating systems they support. For example, an implementation that supports Macintosh shall preserve any OS/2 extended attributes encountered on the media. An implementation that supports Macintosh shall also create and maintain all Macintosh extended attributes specified in this document.
- *Backwards Read Compatibility* – An implementation compliant to this version of the UDF specification *shall* be able to *read* all media written under previous versions of the UDF specification.
- *Backwards Write Compatibility* – UDF 2.xx structures shall not be written to media that contain UDF 1.50 or UDF 1.02 structures. UDF 1.50 and UDF 1.02 structures shall not be written to media that contain UDF 2.xx structures. These two requirements prevent media from containing different versions of the UDF structures.

1.3 General References

1.3.1 References

ISO 9660:1988	Information Processing - Volume and File Structure of CD-ROM for Information Interchange
IEC 908:1987	Compact disc digital audio system
ISO/IEC 10149:1993	Information technology - Data Interchange on Read-Only 120mm optical data discs (CD-ROM based on the Philips/Sony "Yellow Book")
Orange Book part-II	Recordable Compact Disc System Part-II, N.V. Philips and Sony Corporation
Orange Book part-III	Recordable Compact Disc System Part-III, N.V. Philips and Sony Corporation
ISO/IEC 13346:1995	Volume and File Structure of Write-Once and Rewritable media using non-sequential recording for information interchange. This ISO standard is equivalent to ECMA 167 2nd edition.
ECMA 167	ECMA 167 3rd edition is an update to ECMA 167 2nd edition that adds the support for multiple data stream files, and is available from http://www.ecma-international.org/. The previous edition of ECMA 167 (2nd) was is equivalent to ISO/IEC 13346:1995. References enclosed in [ ] in this document are references to ECMA 167 3rd edition. The references are in the form [x/a.b.c], where x is the section number and a.b.c is the paragraph or figure number.

1.3.2 Definitions

Audio session	Audio session contains one or more audio tracks, and no data track.
Audio track	Audio tracks are tracks that are designated to contain audio sectors specified in ISO/IEC 908.
CD-R	CD-Recordable. A Write-Once CD defined in Orange Book, part-II.
CD-RW	CD-Rewritable. An Overwritable CD defined in Orange Book, part-III.
Clean File System	The file system on the media conforms to this specification.
Data track	Data tracks are tracks that are designated to contain data sectors specified in ISO/IEC 10149.
Dirty File System	A file system that is not a clean file system.
ECC Block Size (bytes)	This term refers to values defined in relevant device and/or media specifications. The reader should consult the appropriate document – for example, the "MMC" or "Mt. Fuji" specifications for CD/DVD class media. For media exposing no such concept externally (e.g. hard disc) this term shall be interpreted to mean the sector size of the media.
Fixed Packet	An incremental recording method in which all packets in a given track are of a length specified in the Track Descriptor Block. Addresses presented to a CD drive are translated according to the Method 2 addressing specified in Orange Book parts-II and -III.
ICB	A control node in ECMA 167.

Logical Block Address	A logical block number [3/8.8.1].
	NOTE 1: This is not to be confused with a logical block address [4/7.1], given by the lb_addr structure that contains both a logical block number [3/8.8.1] and a partition reference number [3/8.8], the latter identifying the partition [3/8.7] which contains the addressed logical block [3/8.8.1].
	NOTE 2: A logical block number [3/8.8.1] translates to a logical sector number [3/8.1.2] according to the scheme indicated by the Partition Map [3/10.7] of the partition [3/8.7], which contains the addressed logical block [3/8.8.1]
Media Block Address	A sector number [3/8.1.1], derived from the unique sector address given by a relevant standard for recording [1/5.10]. In this specification, a sector number [3/8.1.1] is equivalent to a logical sector number [3/8.1.2].
Packet	A recordable unit, which is an integer number of contiguous sectors [1/5.9], which consist of user data sectors, and may include additional sectors [1/5.9] which are recorded as overhead of the Packet-writing operation and are addressable according to the relevant standard for recording [1/5.10].
Physical Address	A sector number [3/8.1.1], derived from the unique sector address given by a relevant standard for recording [1/5.10]. In this specification, a sector number [3/8.1.1] is equivalent to a logical sector number [3/8.1.2].
Physical Block Address	A sector number [3/8.1.1], derived from the unique sector address given by a relevant standard for recording [1/5.10]. In this specification, a sector number [3/8.1.1] is equivalent to a logical sector number [3/8.1.2].
physical sector	A sector [1/5.9] given by a relevant standard for recording [1/5.10]. In this specification, a sector [1/5.9] is equivalent to a logical sector [3/8.1.2].
Pseudo OverWrite	Overwrite performed logically by drive on Write-Once media using sequential recording.
Random Access File System	A file system for randomly writable media, either Write-Once or Rewritable
reserved track	A *reserved* track is a track that has a valid Next Writable Address (NWA). For Pseudo OverWrite, this means that sequential write at the NWA and pseudo overwrite until the NWA is possible for this track. See also *used* track.
Sequential File System	A file system for sequentially written media (e.g. CD-R)
Session	The tracks of a volume shall be organized into one or more sessions, e.g. for CD see the Orange Book part-II. A session shall be a sequence of one or more tracks, the track numbers of which form a contiguous ascending sequence.
Track	The sectors of a volume shall be organized into one or more tracks. A track shall be a sequence of sectors, the sector numbers of which form a contiguous ascending sequence. No sector shall belong to more than one track.
	NOTE: There may be gaps between tracks; that is, the last sector of a track need not be adjacent to the first sector of the next track.
UDF	OSTA Universal Disk Format
used track	A *used* track is a track that does not have a valid Next Writable Address. For Pseudo OverWrite, this means that sequential write to this track is not possible. Pseudo overwrite is still possible. See also *reserved* track.

user data blocks The logical blocks [3/8.8.1] which were recorded in the sectors [1/5.9] (equivalent in this specification to logical sectors [3/8.1.2]) of a Packet and which contain the data intentionally recorded by the user of the drive. This specifically does not include the logical blocks [3/8.8.1], if any, whose constituent sectors [1/5.9] were used for the overhead of recording the Packet, even though those sectors [1/5.9] are addressable according to the relevant standard for recording [1/5.10]. Like any logical blocks [3/8.8.1], user data blocks are identified by logical block numbers [3/8.8.1].

user data sectors The sectors [1/5.9] of a Packet which contain the data intentionally recorded by the user of the drive, specifically not including those sectors [1/5.9] used for the overhead of recording the Packet, even though those sectors [1/5.9] may be addressable according to the relevant standard for recording [1/5.10]. Like any sectors [1/5.9], user data sectors are identified by sector numbers [3/8.1.1]. In this specification, a sector number [3/8.1.1] is equivalent to a logical sector number [3/8.1.2].

Variable Packet An incremental recording method in which each packet in a given track is of a host determined length. Addresses presented to a CD drive are as specified in Method 1 addressing in Orange Book parts II and III.

Virtual Address A logical block number [3/8.8.1] of a logical block [3/8.8.1] in a virtual partition. Such a logical block [3/8.8.1] is recorded using the space of a logical block [3/8.8.1] of a corresponding non-virtual partition. The Nth Uint32 in the VAT represents the logical block number [3/8.8.1] in a non-virtual partition used to record logical block number N of its corresponding virtual partition. The first virtual address is 0.

virtual partition A partition of a logical volume [3/8.8] identified in a logical volume descriptor [3/10.6] by a Type 2 Partition Map [3/10.7.3] recorded according section 2.2.8 of this specification. The Virtual Partition Map contains a partition number that is the same as the partition number [3/10.7.2.4] in a Type 1 Partition Map [3/10.7.2] in the same logical volume descriptor [3/10.6]. Each logical block [3/8.8.1] in the virtual partition is recorded using the space of a logical block [3/8.8.1] of that corresponding non-virtual partition. A VAT lists the logical blocks [3/8.8.1] of the non-virtual partition, which have been used to record the logical blocks [3/8.8.1] of its corresponding virtual partition.

virtual sector A logical block [3/8.8.1] in a virtual partition. Such a logical block [3/8.8.1] is recorded using the space of a logical block [3/8.8.1] of a corresponding non-virtual partition. A virtual sector should not be confused with a sector [1/5.9] or a logical sector [3/8.1.2].

VAT A file [4/8.8] recorded in the space of a non-virtual partition which has a corresponding virtual partition, and whose data space [4/8.8.2] is structured according to section 2.2.11 of this specification. This file provides an ordered list of Uint32s, where the Nth Uint32 represents the logical block number [3/8.8.1] of a non-virtual partition used to record logical block number N of its corresponding virtual partition. This file [4/8.8] is not necessarily referenced by a File Identifier Descriptor [4/14.4] of a directory [4/8.6] in the file set [4/8.5] of the logical volume [3/8.8].

VAT ICB A File Entry ICB that describes a file containing a Virtual Allocation Table.

1.3.3 Terms

May	Indicates an action or feature that is optional.
Optional	Describes a feature that may or may not be implemented. If implemented, the feature shall be implemented as described.
Shall	Indicates an action or feature that is mandatory and must be implemented to claim compliance to this standard.
Should	Indicates an action or feature that is optional, but its implementation is strongly recommended.
Reserved	A reserved field is reserved for future use and shall be set to zero. A reserved value is reserved for future use and shall not be used.

1.3.4 Acronyms

Acronym	Definition
AD	Allocation Descriptor
AVDP	Anchor Volume Descriptor Pointer
EA	Extended Attribute
EFE	Extended File Entry
FE	File Entry
FID	File Identifier Descriptor
FSD	File Set Descriptor
ICB	Information Control Block
IUVD	Implementation Use Volume Descriptor
LV	Logical Volume
LVD	Logical Volume Descriptor
LVID	Logical Volume Integrity Descriptor
NWA	Next Writable Address in a track
PD	Partition Descriptor
POW	Pseudo OverWrite as described in appendix 6.15
PVD	Primary Volume Descriptor
SBD	Space Bitmap Descriptor
USD	Unallocated Space Descriptor
VAT	Virtual Allocation Table
VDS	Volume Descriptor Sequence
VRS	Volume Recognition Sequence

2. Basic Restrictions & Requirements

The following table summarizes several of the basic restrictions and requirements defined in this specification. These restrictions & requirements as well as additional ones are described in detail in the following sections of this specification.

Item	Restrictions & Requirements
Logical Sector Size	The *Logical Sector Size* for a specific volume shall be the same as the physical sector size of the specific volume.
Logical Block Size	The *Logical Block Size* for a Logical Volume shall be set to the logical sector size of the volume or volume set on which the specific logical volume resides.
Volume Sets	All media within the same Volume Set shall have the same physical sector size. Rewritable/Overwritable media and WORM media shall not be mixed in/ be present in the same volume set.
First 32K of Volume Space	The first 32768 bytes of the Volume space shall not be used for the recording of ECMA 167 structures. This area shall not be referenced by the Unallocated Space Descriptor or any other ECMA 167 descriptor. This is intended for use by the native operating system.
Volume Recognition Sequence	The Volume Recognition Sequence as described in part 2 of ECMA 167 shall be recorded.
Timestamp	All timestamps shall be recorded in local time. Time zones shall be recorded on operating systems that support the concept of a time zone.
Entity Identifiers	Entity Identifiers shall be recorded in accordance with this document. Unless otherwise specified in this specification the Entity Identifiers shall contain a value that uniquely identifies the implementation.
Descriptor CRCs	CRCs shall be supported and calculated for all Descriptors. There are exception rules for the Descriptor CRC Length of the Space Bitmap Descriptor and the Allocation Extent Descriptor.
File Name Length	Maximum of 255 bytes
Extent Length	Maximum Extent Length shall be $2^{30} - 1$ rounded down to the nearest integral multiple of the Logical Block Size. Maximum Extent Length for extents in virtual space shall be the Logical Block Size.
Primary Volume Descriptor	There shall be exactly one prevailing Primary Volume Descriptor recorded per volume. The media where the *VolumeSequenceNumber* of this descriptor is equal to 1 (one) must be part of the logical volume defined by the prevailing Logical Volume Descriptor.
Anchor Volume Descriptor Pointer	Shall be recorded in at least 2 of the following 3 locations: 256, N-256, or N, where N is the last addressable sector of a volume. See also 2.2.3.

Item	Restrictions & Requirements
Partition Descriptor	A Partition Descriptor Access Type of read-only, rewritable, overwritable, write-once and pseudo-overwritable shall be supported. There shall be exactly one prevailing Partition Descriptor recorded per volume, with one exception. For Volume Sets that consist of single volume, the volume may contain 2 non-overlapping Partitions with 2 prevailing Partition Descriptors only if one has an Access Type of read-only and the other has an Access Type of rewritable, overwritable, or write-once. The Logical Volume for this volume would consist of the contents of both partitions.
Logical Volume Descriptor	There shall be exactly one prevailing Logical Volume Descriptor recorded per Volume Set. The *LogicalVolumeIdentifier* field shall not be null and should contain an identifier that aids in the identification of the logical volume. Specifically, software generating volumes conforming to this specification shall not set this field to a fixed or trivial value. Duplicate disks, which are intended to be identical, may contain the same value in this field. This field is extremely important in logical volume identification when multiple media are present within a jukebox. This name is typically what is displayed to the user. The *Logical Volume Descriptor* recorded on the volume where the *Primary Volume Descriptor*'s *VolumeSequenceNumber* field is equal to 1 (one) must have a *NumberofPartitionMaps* value and *PartitionMaps* structure(s) that represent the entire logical volume. For example, if a volume set is extended by adding partitions, then the updated *Logical Volume Descriptor* written to the last volume in the set must also be written (or rewritten) to the first volume of the set.
Logical Volume Integrity Descriptor	Shall be recorded. The Logical Volume Integrity Sequence extent of LVIDs may be terminated by the extent length.
Partition Integrity Entry	Shall not be recorded, see 2.3.9.
Unallocated Space Descriptor	A single prevailing Unallocated Space Descriptor shall be recorded per volume.
File Set Descriptor	There shall be exactly one File Set Descriptor recorded per Logical Volume. The sole exception is for non-sequential Write-Once media (WORM), see 2.3.2. The FSD extent may be terminated by the extent length.
ICB Tag	Only ICB Strategy Types 4 or 4096 shall be recorded.
File Identifier Descriptor	The total length of a *File Identifier Descriptor* shall not exceed the size of one Logical Block.
File Entry	The total length of a *File Entry* shall not exceed the size of one Logical Block.
Allocation Descriptors	Only Short and Long Allocation Descriptors shall be recorded.
Allocation Extent Descriptors	The length of any single extent of allocation descriptors shall not exceed the *Logical Block Size*.
Unallocated Space Entry	The total length of an *Unallocated Space Entry* shall not exceed the size of one Logical Block.

Item	Restrictions & Requirements
Volume Descriptor Sequence Extent	Both the main and reserve volume descriptor sequence extents shall each have a minimum length of 16 logical sectors. The VDS Extent may be terminated by the extent length.
Record Structure	Record structure files, as defined in part 5 of ECMA 167, shall not be created.
Minimum UDF Read Revision	The Minimum UDF Read Revision value shall be at most #0250 for all media with a UDF 2.60 file system. This indicates that a UDF 2.50 implementation can read all UDF 2.60 media. Media that do not have a Metadata Partition may use a value lower than #250.

2.1 Part 1 - General

2.1.1 Character Sets

The character set used by UDF for the structures defined in this document is the CS0 character set. The OSTA CS0 character set is defined as follows:

OSTA CS0 shall consist of the d-characters specified in The Unicode Standard, Version 2.0 (ISBN 0-201-48345-9 from Addison-Wesley Publishing Company http://www.awl.com/, see also http://www.unicode.org/), excluding #FEFF and #FFFE, stored in the *OSTA Compressed Unicode* format which is defined as follows:

***OSTA Compressed Unicode* format**

RBP	Length	Name	Contents
0	1	Compression ID	Uint8
1	??	Compressed Bit Stream	Byte

The *CompressionID* shall identify the compression algorithm used to compress the *CompressedBitStream* field. The following algorithms are currently supported:

Compression Algorithm

Value	Description
0 - 7	Reserved
8	Value indicates there are 8 bits per character in the *CompressedBitStream*.
9-15	Reserved
16	Value indicates there are 16 bits per character in the *CompressedBitStream*.
17-253	Reserved
254	Value indicates the CS0 expansion is empty and unique. Compression Algorithm 8 is used for compression.
255	Value indicates the CS0 expansion is empty and unique. Compression Algorithm 16 is used for compression.

For a *CompressionID* of 8 or 16, the value of the *CompressionID* shall specify the number of *BitsPerCharacter* for the d-characters defined in the *CharacterBitStream* field. Each sequence of *CompressionID* bits in the *CharacterBitStream* field shall represent an *OSTA Compressed Unicode* d-character. The bits of the character being encoded shall be added to the CharacterBitStream from most- to least-significant-bit. The bits shall be added to the CharacterBitStream starting from the most significant bit of the current byte being encoded into.

NOTE: This encoding causes characters written with a *CompressionID* of 16 to be effectively written in big endian format.

The value of the *OSTA Compressed Unicode* d-character interpreted as a Uint16 defines the value of the corresponding d-character in the Unicode 2.0 standard. Refer to appendix 6.4 on *OSTA Compressed Unicode* for sample C source code to convert between *OSTA Compressed Unicode* and standard Unicode 2.0.

The Unicode byte-order marks, #FEFF and #FFFE, shall not be used.

Compression IDs 254 and 255 shall only be used in FIDs where the Deleted bit is set to ONE.

When uncompressing File Identifiers with Compression IDs 254 and 255, the resulting name is to be considered empty and unique.

2.1.2 OSTA CS0 Charspec

```
struct charspec {          /* ECMA 167 1/7.2.1 */
        Uint8              CharacterSetType;
        byte               CharacterSetInfo[63];
}
```

The *CharacterSetType* field shall have the value of 0 to indicate the CS0 coded character set.

The *CharacterSetInfo* field shall contain the following byte values with the remainder of the field set to a value of 0.

#4F, #53, #54, #41, #20, #43, #6F, #6D, #70, #72, #65, #73, #73, #65, #64, #20, #55, #6E, #69, #63, #6F, #64, #65

The above byte values represent the following ASCII string:
"OSTA Compressed Unicode"

2.1.3 Dstrings

The ECMA 167 standard, as well as this document, has normally defined byte positions relative to 0. In section 1/7.2.12 of ECMA 167, dstrings are defined in terms of being relative to 1. Since this offers an opportunity for confusion, the following shows what the definition would be if described relative to 0.

7.2.12 Fixed-length character fields
A dstring of length *n* is a field of *n* bytes where d-characters (1/7.2) are recorded. The number of bytes used to record the characters shall be recorded as a Uint8 (1/7.1.1) in byte *n-1*, where *n* is

the length of the field. The characters shall be recorded starting with the first byte of the field, and any remaining byte positions after the characters up until byte *n-2* inclusive shall be set to #00.

If the number of d-characters to be encoded is zero, the length of the dstring shall be zero.

NOTE: The length of a dstring includes the compression code byte (2.1.1) except for the case of a zero length string. A zero length string shall be recorded by setting the entire dstring field to all zeros.

2.1.4 Timestamp

```
struct timestamp {          /* ECMA 167 1/7.3 */
    Uint16      TypeAndTimezone;
    Int16       Year;
    Uint8       Month;
    Uint8       Day;
    Uint8       Hour;
    Uint8       Minute;
    Uint8       Second;
    Uint8       Centiseconds;
    Uint8       HundredsofMicroseconds;
    Uint8       Microseconds;
}
```

2.1.4.1 Uint16 TypeAndTimezone;

For the following descriptions *Type* refers to the most significant 4 bits of this field, and *TimeZone* refers to the least significant 12 bits of this field, which is interpreted as a signed 12-bit number in two's complement form.

👓 The time within the structure shall be interpreted as Local Time since *Type* shall be equal to ONE for OSTA UDF compliant media.

✍ *Type* shall be set to ONE to indicate Local Time.

👓 *TimeZone* shall be interpreted as specifying the time zone for the location when this field was last modified. If this field contains -2047 then the time zone has not been specified.

✍ For operating systems that support the concept of a time zone, the offset of the time zone (in 1 minute increments), from Coordinated Universal Time, shall be inserted in the *TimeZone* field. Otherwise the *TimeZone* shall be set to –2047.

NOTE 1: Time zones West of Coordinated Universal Time have negative offsets. For example, Eastern Standard Time is -300 minutes; Eastern Daylight Time is -240 minutes.

NOTE 2: Implementations on systems that support time zones should interpret unspecified time zones as Coordinated Universal Time. Although not a requirement, this interpretation has the advantage that files generated on systems that do not support time zones will always appear to have the same timestamps on systems that do support time zones, irrespective of the interpreting system's local time zone.

2.1.5 Entity Identifier

```
struct EntityID {           /* ECMA 167 1/7.4 */
        Uint8               Flags;
        char                Identifier[23];
        char                IdentifierSuffix[8];
}
```

NOTE: UDF uses *EntityID* for the structure that is called *regid* in ECMA 167.

UDF classifies *Entity Identifiers* into 4 separate types. Each type has its own *Suffix Type* for the *Identifier Suffix* field. The 4 types are:

- *Domain Entity Identifiers* with a *Domain Identifier Suffix*
- *UDF Entity Identifiers* with a *UDF Identifier Suffix*
- *Implementation Entity Identifiers* with an *Implementation Identifier Suffix*
- *Application Entity Identifiers* with an *Application Identifier Suffix*

The following sections describe the format and use of *Entity Identifiers* based upon the different types mentioned above. For all UDF descriptor fields containing an EntityID structure, the value of the *Identifier* field and the *Suffix Type* for the *Identifier Suffix* field are defined in the Entity Identifiers table of 2.1.5.2. The interpretation of the *Identifier Suffix* field for each *Suffix Type* is defined in 2.1.5.3.

2.1.5.1 Uint8 Flags

👓 Self-explanatory.

✍ Shall be set to ZERO.

2.1.5.2 char Identifier[23]

Unless stated otherwise in this document this field shall be set to an identifier that uniquely identifies the implementation. This methodology will allow for identification of the implementation responsible for creating structures recorded on media interchanged between different implementations.
If an implementation updates existing structures on the media written by other implementations the updating implementation shall set the *Identifier* field to a value that uniquely identifies the updating implementation.

The following table summarizes the *Entity Identifier* fields defined in the ECMA 167 standard and this document and shows to what values they shall be set.

Entity Identifiers

Descriptor	Field	ID Value	Suffix Type
Primary Volume Descriptor	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Primary Volume Descriptor	Application ID	*"*Application ID"*	Application Identifier Suffix
Implementation Use Volume Descriptor	Implementation Identifier	"*UDF LV Info"	UDF Identifier Suffix
Implementation Use Volume Descriptor	Implementation ID (in Implementation Use field)	*"*Developer ID"*	Implementation Identifier Suffix
Partition Descriptor	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Partition Descriptor	Partition Contents	*"+NSR03"*	Application Identifier Suffix
Logical Volume Descriptor	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Logical Volume Descriptor	Domain ID	"*OSTA UDF Compliant"	Domain Identifier Suffix
File Set Descriptor	Domain ID	"*OSTA UDF Compliant"	Domain Identifier Suffix
File Identifier Descriptor	Implementation Use (optional)	*"*Developer ID"*	Implementation Identifier Suffix
File Entry	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Device Specification Extended Attribute	Implementation Use	*"*Developer ID"*	Implementation Identifier Suffix
UDF Implementation Use Extended Attribute	Implementation ID	*See 3.3.4.5*	UDF Identifier Suffix
Non-UDF Implementation Use Extended Attribute	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
UDF Application Use Extended Attribute	Application ID	*See 3.3.4.6*	UDF Identifier Suffix
Non-UDF Application Use Extended Attribute	Application ID	*"*Application ID"*	Application Identifier Suffix

UDF Unique ID Mapping Data	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Power Calibration Table Stream	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Logical Volume Integrity Descriptor	Implementation ID (in Implementation Use field)	*"*Developer ID"*	Implementation Identifier Suffix
Partition Integrity Entry	Implementation ID	N/A, see 2.3.9	N/A
Virtual Partition Map	Partition Type Identifier	"*UDF Virtual Partition"	UDF Identifier Suffix
Virtual Allocation Table	Implementation Use (optional)	*"*Developer ID"*	Implementation Identifier Suffix
Sparable Partition Map	Partition Type Identifier	"*UDF Sparable Partition"	UDF Identifier Suffix
Sparing Table	Sparing Identifier	"*UDF Sparing Table"	UDF Identifier Suffix
Metadata Partition Map	Partition Type Identifier	"*UDF Metadata Partition"	UDF Identifier Suffix

The *Suffix Type* column in the above table defines the format of the suffix to be used with the corresponding Entity Identifier. These different suffix types are defined in the following section 2.1.5.3.

NOTE 1: The value of the Entity Identifier field is interpreted as a sequence of bytes, and not as a dstring specified in CS0. For ease of use the values used by UDF for this field are specified in terms of ASCII character strings. The actual sequence of bytes used for the Entity Identifiers defined by UDF are specified in section 6.2.

In the *ID Value* column in the above table *"*Developer ID"* refers to an Entity Identifier that uniquely identifies the current implementation. The value specified should be used when a new descriptor is created. Also, the value specified should be used for an existing descriptor when anything within the scope of the specified EntityID field is modified.

NOTE 2: The value chosen for a *"*Developer ID"* should contain enough information to identify the company and product name for an implementation. For example, a company called *XYZ* with a UDF product called *DataOne* might choose *"*XYZ DataOne"* as their Developer ID. Also in the suffix of their Developer ID they may choose to record the current version number of their *DataOne* product. This information is extremely helpful when trying to determine which implementation wrote a bad structure on a piece of media when multiple products from different companies have been recording on the media.

In the *ID Value* column in the above table "**Application ID"* refers to an identifier that uniquely identifies the writer's application.

NOTE 3: All *Identifiers* defined in this document (appendix 6.1) shall be registered by OSTA as UDF *Identifiers*.

2.1.5.3 char IdentifierSuffix[8]

The format of the *Identifier Suffix* field is dependent on the type of the *Identifier*.

In regard to OSTA *Domain Entity Identifiers* specified in this document (see 2.1.5.2 and appendix 6.1), the *Identifier Suffix* field shall be constructed as follows:

Domain Identifier Suffix format

RBP	Length	Name	Contents
0	2	UDF Revision	Uint16 (= #0260)
2	1	Domain Flags	Uint8
3	5	Reserved	bytes (= #00)

The *UDF Revision* field shall contain **#0260** to indicate revision **2.60** of this document. This field will allow an implementation to detect changes made in newer revisions of this document. The OSTA Domain Identifiers are only used in the Logical Volume Descriptor and the File Set Descriptor. The *DomainFlags* field defines the following bit flags:

Domain Flags

Bit	Description
0	HardWriteProtect
1	SoftWriteProtect
2-7	Reserved

The *SoftWriteProtect* flag is a user settable flag that indicates that the volume or file structures within the scope of the descriptor in which it resides are write protected. A *SoftWriteProtect* flag value of ONE shall indicate user write protected structures. This flag may be set or reset by the user.
The *HardWriteProtect* flag is an implementation settable flag that indicates that the scope of the descriptor in which it resides is permanently write protected. A *HardWriteProtect* flag value of ONE shall indicate a permanently write protected structure. Once set this flag shall not be reset. The *HardWriteProtect* flag overrides the *SoftWriteProtect* flag.

The write protect flags appear in the Logical Volume Descriptor and in the File Set Descriptor. They shall be interpreted as follows:

is_fileset_write_protected = LVD.HardWriteProtect || LVD.SoftWriteProtect ||
    FSD.HardWriteProtect || FSD.SoftWriteProtect
is_fileset_hard_protected = LVD.HardWriteProtect || FSD.HardWriteProtect
is_fileset_soft_protected = (LVD.SoftWriteProtect || FSD.SoftWriteProtect) &&
    !is_fileset_hard_protected
is_vol_write_protected = LVD.HardWriteProtect || LVD.SoftWriteProtect
is_vol_hard_protected = LVD.HardWriteProtect
is_vol_soft_protected = LVD.SoftWriteProtect && !LVD.HardWriteProtect

For *UDF Entity Identifiers* as defined by UDF (see 2.1.5.2 and appendix 6.1), the *Identifier Suffix* field shall be constructed as follows:

UDF Identifier Suffix format

RBP	Length	Name	Contents
0	2	UDF Revision	Uint16 (= #0260)
2	1	OS Class	Uint8
3	1	OS Identifier	Uint8
4	4	Reserved	bytes (= #00)

The contents of the *OS Class* and OS *Identifier* fields are described in Appendix 6.3 on *Operating System Identifiers*.

For *Implementation Entity Identifiers* not defined by UDF (see 2.1.5.2), the *Identifier Suffix* field shall be constructed as follows:

Implementation Identifier Suffix format

RBP	Length	Name	Contents
0	1	OS Class	Uint8
1	1	OS Identifier	Uint8
2	6	Implementation Use Area	bytes

NOTE: It is important to understand the intended use and importance of the *OS Class* and *OS Identifier* fields. The main purpose of these fields is to aid in debugging when problems are found on a UDF volume. The fields also provide useful information that could be provided to the end user. When set correctly these two fields provide an implementation with information such as the following:

- Identify under which operating system a particular structure was last modified.
- Identify under which operating system a specific file or directory was last modified.
- If a developer supports multiple operating systems with their implementation, it helps to determine under which operating system a problem may have occurred.

For an *Application Entity Identifier* not defined by UDF (see 2.1.5.2), the *Identifier Suffix* field shall be constructed as follows, unless specified otherwise.

Application Identifier Suffix format

RBP	Length	Name	Contents
0	8	Application Use Area	bytes

2.1.6 Descriptor Tag Serial Number at Formatting Time

In order to support disaster recovery, the *Tag Serial Number* value of all UDF descriptors that will be recorded at formatting time, shall be set to a value that differs from ones previously recorded, upon volume re-initialization.

If no disaster recovery will be supported, a value zero (#0000) shall be used for the *Tag Serial Number* field of all UDF descriptors that will be recorded at formatting time, see 2.2.1.1, 2.3.1.1 and ECMA 167 3/7.2.5 and 4/7.2.5.

If disaster recovery is supported, the value to be used depends on the state of the volume prior to formatting. There are only two states in which a volume can be formatted such that disaster recovery will be possible in the future. These states are:

1) The volume is completely erased. Only after this action, and where disaster recovery is to be supported then a value of one (#0001) shall be used as the *Tag Serial Number* value.

2) The volume is a clean UDF volume that supports disaster recovery for *Tag Serial Number* values, and the *Tag Serial Number* values of at least two Anchor Volume Descriptor Pointers are both equal to X, where X is not equal to zero. If disaster recovery is to be supported then a value X+1 shall be used as the *Tag Serial Number* value. If X+1 wraps to zero then keep it as zero to indicate that disaster recovery is not supported.

   NOTE 1: The reason for this is that if X+1 wraps to zero then the uniqueness of any *Tag Serial Number* value unequal to zero can no longer be guaranteed on the volume.

NOTE 2: By 'erased' in the above paragraphs, we mean that the sectors are made non-valid for UDF - for example by writing zeroes to the sectors.

2.1.7 Volume Recognition Sequence

The following rules shall apply when writing the volume recognition sequence:

✍ The Volume Recognition Sequence (VRS) as described in part 2 and part 3 of ECMA 167 shall be recorded. There shall be exactly one NSR descriptor in the VRS. The NSR and BOOT2 descriptors shall be in the Extended Area. There shall be only one Extended Area with one BEA01 and one TEA01. All other VSDs are only allowed before the Extended Area. The first sector after the VRS shall be unrecorded or contain all #00 bytes.

☞ Implementers should expect that media recorded by UDF 2.00 and lower revisions do not have the requirement mentioned above concerning the first sector after the VRS.

NOTE: Currently, no BOOT2 descriptor is defined for UDF, see 5.3. Further, see ECMA 167 part 2, 3/3.1, 3/3.2 and 3/9.1.

2.2 Part 3 - Volume Structure

2.2.1 Descriptor Tag

```
struct tag {            /* ECMA 167 3/7.2 */
    Uint16              TagIdentifier;
    Uint16              DescriptorVersion;
    Uint8               TagChecksum;
    byte                Reserved;
    Uint16              TagSerialNumber;
    Uint16              DescriptorCRC;
    Uint16              DescriptorCRCLength;
    Uint32              TagLocation;
}
```

NOTE: The value zero for TagIdentifier is not defined by ECMA 167, but it is used by UDF for the Sparing Table.

2.2.1.1 Uint16 TagSerialNumber

🕶 Ignored. Intended for disaster recovery.

✍ Shall be set to the *Tag Serial Number* value of the Anchor Volume Descriptor Pointers on this volume.

In order to preserve disaster recovery support, the *Tag Serial Number* must be set to a value that differs from ones previously recorded, upon volume re-initialization. This value is determined at volume formatting time and may depend on the state of the volume prior to formatting. See 2.1.6 for further details.

2.2.1.2 Uint16 DescriptorCRCLength

Descriptor CRCs shall be supported and calculated for each descriptor. Unless otherwise specified, the value of the *DescriptorCRCLength* field shall be set to the minimum of the following two values: ((Size of the Descriptor) - (Length of Descriptor Tag)); 65535. When reading a descriptor, the Descriptor CRC should be validated.

NOTE: The *DescriptorCRCLength* field must not be used to determine the actual length of the descriptor or the number of bytes to be read. These lengths do not match in all cases because of possible *DescriptorCRCLength* truncation to 65535 and other *DescriptorCRCLength* exceptions as specified in this standard.

2.2.2 Primary Volume Descriptor

```
struct PrimaryVolumeDescriptor {                /* ECMA 167 3/10.1 */
        struct tag          DescriptorTag;
        Uint32              VolumeDescriptorSequenceNumber;
        Uint32              PrimaryVolumeDescriptorNumber;
        dstring             VolumeIdentifier[32];
        Uint16              VolumeSequenceNumber;
        Uint16              MaximumVolumeSequenceNumber;
        Uint16              InterchangeLevel;
        Uint16              MaximumInterchangeLevel;
        Uint32              CharacterSetList;
        Uint32              MaximumCharacterSetList;
        dstring             VolumeSetIdentifier[128];
        struct charspec     DescriptorCharacterSet;
        struct charspec     ExplanatoryCharacterSet;
        struct extent_ad    VolumeAbstract;
        struct extent_ad    VolumeCopyrightNotice;
        struct EntityID     ApplicationIdentifier;
        struct timestamp    RecordingDateandTime;
        struct EntityID     ImplementationIdentifier;
        byte                ImplementationUse[64];
        Uint32              PredecessorVolumeDescriptorSequenceLocation;
        Uint16              Flags;
        byte                Reserved[22];
}
```

2.2.2.1 Uint16 InterchangeLevel

Interpreted as specifying the current interchange level (as specified in ECMA 167 3/11), of the contents of the associated volume and the restrictions implied by the specified level.

If this volume is part of a multi-volume Volume Set then the level shall be set to 3, otherwise the level shall be set to 2.

ECMA 167 requires an implementation to enforce the restrictions associated with the specified current *Interchange Level*. The implementation may change the value of this field as long as it does not exceed the value of the *Maximum Interchange Level* field.

2.2.2.2 Uint16 MaximumInterchangeLevel

Interpreted as specifying the maximum interchange level (as specified in ECMA 167 3/11), of the contents of the associated volume.

This field shall be set to level 3 (No Restrictions Apply), unless specifically given a different value by the user.

NOTE: This field is used to determine the intent of the originator of the volume. If this field has been set to 2 then the originator does not wish the volume to be included in a multi-volume set (interchange level 3). The receiver may override this field and set it to a 3 but the implementation should give the receiver a strict warning explaining the intent of the originator of the volume.

2.2.2.3 Uint32 CharacterSetList

👓 Interpreted as specifying the character set(s) in use by any of the structures defined in Part 3 of ECMA 167 (3/10.1.9).

✍ Shall be set to indicate support for CS0 only as defined in 2.1.2.

2.2.2.4 Uint32 MaximumCharacterSetList

👓 Interpreted as specifying the maximum supported character sets (as specified in ECMA 167) which may be specified in the *CharacterSetList* field.

✍ Shall be set to indicate support for CS0 only as defined in 2.1.2.

2.2.2.5 dstring VolumeSetIdentifier[128]

👓 Interpreted as specifying the identifier for the volume set .

✍ The first 16 characters of this field should be set to a unique value. The remainder of the field may be set to any allowed value. Specifically, software generating volumes conforming to this specification shall not set this field to a fixed or trivial value. Duplicate disks which are intended to be identical may contain the same value in this field.

NOTE: The intended purpose of this is to guarantee Volume Sets with unique identifiers. The first 8 characters of the unique part should come from a CS0 hexadecimal representation of a 32-bit time value. The remaining 8 characters are free for implementation use.

2.2.2.6 struct charspec DescriptorCharacterSet

👓 Interpreted as specifying the character sets allowed in the *Volume Identifier* and *Volume Set Identifier* fields.

✍ Shall be set to indicate support for CS0 as defined in 2.1.2.

2.2.2.7 struct charspec ExplanatoryCharacterSet

👓 Interpreted as specifying the character sets used to interpret the contents of the *VolumeAbstract* and *VolumeCopyrightNotice* extents.

✍ Shall be set to indicate support for CS0 as defined in 2.1.2.

2.2.2.8 struct EntityID ImplementationIdentifier

For more information on the proper handling of this field see section 2.1.5.

2.2.2.9 struct EntityID ApplicationIdentifier

☞ This field either specifies a valid Entity Identifier (section 2.1.5) identifying the application that last wrote this field, or the field is filled with all #00 bytes, meaning that no application is identified.

✍ Either all #00 bytes or a valid Entity Identifier (section 2.1.5) shall be recorded in this field.

2.2.3 Anchor Volume Descriptor Pointer

```
struct AnchorVolumeDescriptorPointer {        /* ECMA 167 3/10.2 */
        struct tag          DescriptorTag;
        struct extent_ad    MainVolumeDescriptorSequenceExtent;
        struct extent_ad    ReserveVolumeDescriptorSequenceExtent;
        byte                Reserved[480];
}
```

NOTE 1: An *Anchor Volume Descriptor Pointer* structure shall be recorded in at least 2 of the following 3 locations on the media:

- Logical Sector 256.
- Logical Sector (N - 256).
- N

NOTE 2: Closed media shall conform to the above rules. As specified in section 6.11.2, unclosed sequential Write-Once media may have a single AVDP present at either sector 256 or 512. If on an unclosed disc a single AVDP is recorded on sector 256, any AVDP recorded on sector 512 must be ignored.

2.2.3.1 struct MainVolumeDescriptorSequenceExtent

The *Main Volume Descriptor Sequence Extent* shall have a minimum length of 16 logical sectors.

2.2.3.2 struct ReserveVolumeDescriptorSequenceExtent

The *Reserve Volume Descriptor Sequence Extent* shall have a minimum length of 16 logical sectors.

NOTE: The Main VDS extent and the Reserve VDS extent shall be recorded in different ECC blocks. The locations of these extents on the volume should be as far apart as physically possible. Typically this is achieved by maximizing the difference between the start LSNs of the extents. Care should be taken in case of special LSN address schemes, e.g. for multiple layer media.

2.2.4 Logical Volume Descriptor

```
struct LogicalVolumeDescriptor {                /* ECMA 167 3/10.6 */
        struct tag          DescriptorTag;
        Uint32              VolumeDescriptorSequenceNumber;
        struct charspec     DescriptorCharacterSet;
        dstring             LogicalVolumeIdentifier[128];
        Uint32              LogicalBlockSize,
        struct EntityID     DomainIdentifier;
        byte                LogicalVolumeContentsUse[16];
        Uint32              MapTableLength;
        Uint32              NumberofPartitionMaps;
        struct EntityID     ImplementationIdentifier;
        byte                ImplementationUse[128];
        extent_ad           IntegritySequenceExtent,
        byte                PartitionMaps[];
}
```

2.2.4.1 struct charspec DescriptorCharacterSet

- 🕶 Interpreted as specifying the character set allowed in the *LogicalVolumeIdentifier* field.

- ✍ Shall be set to indicate support for CS0 as defined in 2.1.2.

2.2.4.2 Uint32 LogicalBlockSize

- 🕶 Interpreted as specifying the *Logical Block Size* for the logical volume identified by this *Logical Volume Descriptor*.

- ✍ This field shall be set to the largest logical sector size encountered amongst all the partitions on media that constitute the logical volume identified by this *Logical Volume Descriptor*. Since UDF requires that all Volumes within a Volume Set have the same logical sector size, the *Logical Block Size* will be the same as the logical sector size of the Volume.

2.2.4.3 struct EntityID DomainIdentifier

- 🕶 Interpreted as specifying a domain specifying rules on the use of, and restrictions on, certain fields in the descriptors. If this field is all zero then it is ignored, otherwise the *Entity Identifier* rules are followed.

  NOTE 1: If the field does not contain "***OSTA UDF Compliant**" then an implementation may deny the user access to the logical volume.

✍ This field shall indicate that the contents of this logical volume conforms to the domain defined in this document, therefore the *Domain Identifier* ID value shall be set to:

"*OSTA UDF Compliant"

As described in the section on *Entity Identifier* the *Identifier Suffix* field of this *EntityID* shall contain the revision of this document for which the contents of the Logical Volume is compatible. For more information on the proper handling of this field see section 2.1.5.

NOTE 2: The *Identifier Suffix* field of this EntityID contains *SoftWriteProtect* and *HardWriteProtect* flags. Refer to 2.1.5.3.

2.2.4.4 byte LogicalVolumeContentsUse[16]

This field contains the extent location of the File Set Descriptor. This is described in 4/3.1 of ECMA 167 as follows:

> "If the volume is recorded according to Part 3, the extent in which the first File Set Descriptor Sequence of the logical volume is recorded shall be identified by a long_ad (4/14.14.2) recorded in the Logical Volume Contents Use field (see 3/10.6.7) of the Logical Volume Descriptor describing the logical volume in which the File Set Descriptors are recorded."

This field can be used to find the File Set Descriptor, and from the File Set Descriptor the root directory can be found.

2.2.4.5 struct EntityID ImplementationIdentifier;

For more information on the proper handling of this field see section 2.1.5.

2.2.4.6 struct extent_ad IntegritySequenceExtent

A value in this field is required for the Logical Volume Integrity Descriptor. For Rewriteable or Overwriteable media this shall be set to a minimum of 8K bytes.

WARNING: *For WORM media this field should be set to an extent of some substantial length.* Once the WORM volume on which the Logical Volume Integrity Descriptor resides is full a new volume must be added to the volume set since the Logical Volume Integrity Descriptor must reside on the same volume as the prevailing Logical Volume Descriptor.

2.2.4.7 byte PartitionMaps[]

For the purpose of interchange, Partition Maps shall be limited to Partition Map type 1, except type 2 maps as described in this document (2.2.8, 2.2.9 and 2.2.10).

2.2.5 Unallocated Space Descriptor

```
struct UnallocatedSpaceDesc {                    /* ECMA 167 3/10.8 */
        struct tag          DescriptorTag;
        Uint32              VolumeDescriptorSequenceNumber;
        Uint32              NumberofAllocationDescriptors;
        extent_ad           AllocationDescriptors[];
}
```

This descriptor shall be recorded, even if there is no free volume space. The first 32768 bytes of the Volume space shall not be used for the recording of ECMA 167 structures. This area shall not be referenced by the Unallocated Space Descriptor or any other ECMA 167 descriptor.

2.2.6 Logical Volume Integrity Descriptor

```
struct LogicalVolumeIntegrityDesc {              /* ECMA 167 3/10.10 */
        struct tag          DescriptorTag,
        Timestamp           RecordingDateAndTime,
        Uint32              IntegrityType,
        struct extent_ad    NextIntegrityExtent,
        byte                LogicalVolumeContentsUse[32],
        Uint32              NumberOfPartitions,
        Uint32              LengthOfImplementationUse,   /* = L_IU */
        Uint32              FreeSpaceTable[],
        Uint32              SizeTable[],
        byte                ImplementationUse[]
}
```

The *Logical Volume Integrity Descriptor* is a structure that shall be written any time the contents of the associated Logical Volume is modified. Through the contents of the *Logical Volume Integrity Descriptor* an implementation can easily answer the following useful questions:

1) Are the contents of the Logical Volume in a consistent state?
2) When was the last date and time that anything within the Logical Volume was modified?
3) What is the total Logical Volume free space in logical blocks?
4) What is the total size of the Logical Volume in logical blocks?
5) What is the next available UniqueID for use within the Logical Volume?
6) Has some *other* implementation modified the contents of the logical volume since the last time that the *original* implementation, which created the logical volume, accessed it.

2.2.6.1 byte LogicalVolumeContentsUse[32]

See section 3.2.1 for information on the contents of this field.

2.2.6.2 Uint32 FreeSpaceTable[]

Since most operating systems require that an implementation provides the true free space of a Logical Volume at mount time it is important that the Free Space Table values be maintained for all partitions, except for the following two cases:

1. For a virtual partition and for a partition with Access Type pseudo-overwritable, the Free Space Table value shall be set to #FFFFFFFF.
2. For a partition with Access Type read-only, the Free Space Table value shall be set to zero.

In all other cases, the optional value of #FFFFFFFF, which indicates that the amount of available free space is not known, shall not be used.

NOTE: The Free Space Table is guaranteed to be correct only when the *Logical Volume Integrity Descriptor* is closed.

2.2.6.3 Uint32 SizeTable[]

Since most operating systems require that an implementation provide the total size of a Logical Volume at mount time it is important that these values be maintained for all non-virtual partitions. The optional value of #FFFFFFFF, which indicates that the partition size is not known, shall not be used for non-virtual partitions. For virtual partitions the SizeTable value shall be set to #FFFFFFFF.

2.2.6.4 byte ImplementationUse[]

The *ImplementationUse* area for the *Logical Volume Integrity Descriptor* shall be structured as follows:

ImplementationUse format

RBP	Length	Name	Contents
0	32	ImplementationID	EntityID
32	4	Number of Files	Uint32
36	4	Number of Directories	Uint32
40	2	Minimum UDF Read Revision	Uint16
42	2	Minimum UDF Write Revision	Uint16
44	2	Maximum UDF Write Revision	Uint16
46	L_IU - 46	Implementation Use	byte

NOTE: For a Sequential File System using a VAT, all field values above will be overruled by the corresponding VAT fields, except for the ImplementationID and Implementation Use fields, see 2.2.11.

Implementation ID - The implementation identifier *EntityID* of the implementation which last modified anything within the scope of this *EntityID*. The scope of this *EntityID* is the Logical Volume Descriptor, and the contents of the associated Logical Volume. This field allows an implementation to identify which implementation last modified the contents of a Logical Volume.

Number of Files - The current number of files in the Logical Volume, including hard links. The count includes all FIDs in the directory hierarchy for which the Directory bit, Parent bit and Deleted bit are all ZERO. FIDs identifying a Named Stream are not included in the count. This information is needed by the Macintosh OS. All implementations shall maintain this information.

Number of Directories - The current number of directories in the Logical Volume, plus the root directory. The count includes the root directory and all FIDs in the directory hierarchy for which the Directory bit is ONE and the Parent bit and Deleted bit are both ZERO. FIDs identifying a Stream Directory are not included in the count. This information is needed by the Macintosh OS. All implementations shall maintain this information.

Minimum UDF Read Revision - Shall indicate the minimum recommended revision of the UDF specification that an implementation is required to support to successfully be able to read all potential structures on the media. This number shall be stored in binary coded decimal format, for example #0250 would indicate revision 2.50 of the UDF specification. See further requirements in the Basic Restrictions & Requirements section.

Minimum UDF Write Revision - Shall indicate the minimum revision of the UDF specification that an implementation is required to support to successfully be able to modify all structures on the media. This number shall be stored in binary coded decimal format, for example #0250 would indicate revision 2.50 of the UDF specification.

Maximum UDF Write Revision - Shall indicate the maximum revision of the UDF specification that an implementation that has modified the media has supported. An implementation shall update this field only if it has modified the media and the level of the UDF specification it supports is higher than the current value of this field. This number shall be stored in binary coded decimal format, for example #0260 would indicate revision 2.60 of the UDF specification.

Implementation Use - Contains implementation specific information unique to the implementation identified by the Implementation ID.

2.2.7 Implementation Use Volume Descriptor

```
struct ImpUseVolumeDescriptor {                /* ECMA 167 3/10.4 */
        struct tag          DescriptorTag;
        Uint32              VolumeDescriptorSequenceNumber;
        struct EntityID     ImplementationIdentifier;
        byte                ImplementationUse[460];
}
```

This section defines an UDF Implementation Use Volume Descriptor. This descriptor shall be recorded on every Volume of a Volume Set. The Volume may also contain additional Implementation Use Volume Descriptors that are implementation specific. The intended purpose of this descriptor is to aid in the identification of a Volume within a Volume Set that belongs to a specific Logical Volume.

NOTE: An implementation may still record an additional Implementation Use Volume Descriptor in its own format on the media. The UDF Implementation Use Volume Descriptor does not preclude an additional descriptor.

2.2.7.1 EntityID ImplementationIdentifier

The Identifier field of this EntityID shall specify "***UDF LV Info**". Refer to section 2.1.5 on Entity Identifier.

2.2.7.2 bytes ImplementationUse[460]

The implementation use area shall contain the following structure:

```
struct LVInformation {
        struct charspec     LVICharset,
        dstring             LogicalVolumeIdentifier[128],
        dstring             LVInfo1[36],
        dstring             LVInfo2[36],
        dstring             LVInfo3[36],
        struct EntityID     ImplementationID,
        bytes               ImplementationUse[128];
}
```

2.2.7.2.1 charspec LVICharset

- 👓 Interpreted as specifying the character sets allowed in the *LogicalVolumeIdentifier* and *LVInfo* fields.
- ✍ Shall be set to indicate support for CS0 only as defined in 2.1.2.

2.2.7.2.2 dstring LogicalVolumeIdentifier[128]

Identifies the Logical Volume referenced by this descriptor.

2.2.7.2.3 dstring LVInfo1[36], LVInfo2[36] and LVInfo3[36]

The fields LVInfo1, LVInfo2 and LVInfo3 should contain additional information to aid in the identification of the media. For example the LVInfo fields could contain information such as *Owner Name*, *Organization Name*, and *Contact Information*.

2.2.7.2.4 struct EntityID ImplementationID

Refer to section 2.1.5 on Entity Identifier.

2.2.7.2.5 bytes ImplementationUse[128]

This area may be used by the implementation to store any additional implementation specific information.

2.2.8 Virtual Partition Map

This is an extension of ECMA 167 to expand its scope to include sequentially written media (eg. CD-R). This extension is for a Partition Map entry to describe a virtual space.

The Logical Volume Descriptor contains a list of partitions that make up a given volume. As the virtual partition cannot be described in the same manner as a physical partition, a Type 2 Partition Map defined below shall be used.

If a Virtual Partition Map is recorded, then the Logical Volume Descriptor shall contain at least two Partition Maps. One Partition Map shall be recorded as a Type 1 Partition Map. One Partition Map shall be recorded as a Type 2 Partition Map. The format of this Type 2 Partition Map shall be as specified in the following table.

Layout of Type 2 Partition Map for virtual partition

RBP	Length	Name	Contents
0	1	Partition Map Type	Uint8 = 2
1	1	Partition Map Length	Uint8 = 64
2	2	Reserved 1	#00 bytes
4	32	Partition Type Identifier	EntityID
36	2	Volume Sequence Number	Uint16
38	2	Partition Number	Uint16
40	24	Reserved 2	#00 bytes

- Partition Type Identifier:
  - Flags = 0
  - Identifier = `*UDF Virtual Partition`
  - Identifier Suffix is recorded as defined in section 2.1.5
- Volume Sequence Number = volume upon which the VAT and Partition is recorded
- Partition Number = the partition number in the Type 1 Partition Map in the same logical volume descriptor.

2.2.9 Sparable Partition Map

Certain disk/drive systems do not perform defect management (eg. CD-RW). To provide an apparent defect-free space for these systems, a partition of type 2 is used. The Partition Map defines the partition number, packet size (see section 1.3.2), and size and locations of the Sparing Tables. This type 2 map is intended to replace the type 1 map normally found on the media. There should not be a type 1 map recorded if a Sparable Partition Map is recorded. The Sparable Partition Map identifies not only the partition number and the volume sequence number, but also identifies the Packet Length and the Sparing Tables. A Sparable Partition Map shall not be recorded on disk/drive systems that perform defect management.

Layout of Type 2 Partition Map for sparable partition

RBP	Length	Name	Contents
0	1	Partition Map Type	Uint8 = 2
1	1	Partition Map Length	Uint8 = 64
2	2	Reserved 1	#00 bytes
4	32	Partition Type Identifier	EntityID
36	2	Volume Sequence Number	Uint16
38	2	Partition Number	Uint16
40	2	Packet Length	Uint16
42	1	Number of Sparing Tables (=N_ST)	Uint8
43	1	Reserved 2	#00 byte
44	4	Size of each Sparing Table	Uint32
48	4 * N_ST	Locations of Sparing Tables	Uint32
48 + 4 * N_ST	16 - 4 * N_ST	Pad	#00 bytes

- Partition Type Identifier:
  - Flags = 0
  - Identifier = `*UDF Sparable Partition`
  - Identifier Suffix is recorded as defined in section 2.1.5.
- Partition Number = the number of this partition. Shall identify a Partition Descriptor associated with this partition.
- Packet Length = the number of user data blocks per fixed packet. This value is specified in the medium specific sections of the Appendices in section 6.
- Number of Sparing Tables = the number of redundant tables recorded. This shall be a value in the range of 1 to 4.
- Size of each Sparing Table = Length, in bytes, allocated for each Sparing Table.
- Locations of Sparing Tables = the start locations of each Sparing Table specified as a media block address. Implementations should align the start of each Sparing Table with the beginning of a packet. Implementations should record at least two Sparing Tables in physically distant locations.

2.2.10 Metadata Partition Map

A Metadata Partition Map *shall* be recorded for volumes that contain a single Partition Descriptor having an Access Type of 0 (pseudo-overwritable), 1 (read-only) or 4 (overwritable) and do not have a Virtual Partition Map recorded in the LVD. For the special case of two non-overlapping Partitions on one volume, one with an Access Type of read-only and one with an Access Type overwritable, there shall be a Metadata Partition Map associated with the overwritable partition.
A Metadata Partition Map *shall not* be recorded in all other cases.

See section 2.2.13 for further description of the Metadata Partition.

Layout of Type 2 Partition Map for metadata partition

RBP	Length	Name	Contents
0	1	Partition Map Type	Uint8 = 2
1	1	Partition Map Length	Uint8 = 64
2	2	Reserved 1	#00 bytes
4	32	Partition Type Identifier	EntityID
36	2	Volume Sequence Number	Uint16
38	2	Partition Number	Uint16
40	4	Metadata File Location	Uint32
44	4	Metadata Mirror File Location	Uint32
48	4	Metadata Bitmap File Location	Uint32
52	4	Allocation Unit Size (blocks)	Uint32
56	2	Alignment Unit Size (blocks)	Uint16
58	1	Flags	Uint8
59	5	Reserved 2	#00 bytes

- Partition Type Identifier:
  - Flags = 0
  - Identifier = `*UDF Metadata Partition`
  - Identifier Suffix is recorded as in section 2.1.5.
- Partition Number = the number of this partition. Shall identify a Partition Descriptor associated with this partition. This shall match the partition number in the Type 1 Partition Map or Type 2 Sparable Partition Map, one and only one of which shall also be recorded as appropriate to the media type.
- Metadata File Location = address of the block containing the File Entry for the Metadata File. This address shall be interpreted as a logical block number within the physical or sparable partition associated with this Partition Map (see above "Partition Number" field).
- Metadata Mirror File Location = address of block containing the File Entry for the metadata file mirror. This address shall be interpreted as a logical block number within the physical or sparable partition associated with this Partition Map (see above "Partition Number" field).
- Metadata Bitmap File Location = the address of the block containing the File Entry for the Metadata Bitmap File. This address shall be interpreted as a logical block number within the physical or sparable partition associated with this Partition Map (see above "Partition Number" field). If the value of the Metadata Bitmap File Location field is equal to #FFFFFFFF, no File Entry for the Metadata Bitmap File is defined.
- Allocation Unit Size = the number of logical blocks per Allocation Unit for the metadata file (and mirror file) associated with this Partition Map. This value shall be an integer multiple of the larger of the following three values: (media ECC Block Size (divided by) logical block size); Packet Length (if a type 2 Sparable Partition Map is recorded); 32.
- Alignment Unit Size (blocks) = all extents allocated to the Metadata File (or Mirror File) must have a starting Lbn which is an integer multiple of this value. This value shall be an integer multiple of the larger of the following: (media ECC Block Size (divided by) logical block size); Packet Length (if a type 2 Sparable Partition Map is recorded).
- Flags:
  - Bit 0: "Duplicate Metadata Flag": When set, indicates that the Metadata Mirror File has its own unique allocation (i.e. it duplicates the data in the Metadata File). When clear indicates that the Metadata Mirror File allocation descriptors describe the same allocation as the Metadata File allocation descriptors (i.e. the data is not duplicated, and the data blocks are shared between both main and mirror files, but each File Entry and its associated allocation descriptors are unique and distinct).
  - Bits 1-7: Reserved. Shall be set to zero on write, and ignored on read.

NOTE 1: The Metadata Partition shall have an entry in the LVID Size Table and Free Space Table (see 2.2.6).

NOTE 2: The Metadata File Location, Metadata Mirror File Location and Metadata Bitmap File Location Uint32 fields define File Entry locations. The number of blocks allocated for each File Entry shall be one logical block.

2.2.11 Virtual Allocation Table

The Virtual Allocation Table (VAT) is used on sequentially written media (eg. CD-R) to give the appearance of randomly writable media to the system. The existence of this partition is identified in the Partition Maps. The VAT shall only be recorded on sequentially written media (eg. CD-R).

The VAT is a map that translates Virtual Addresses to logical addresses. It shall be recorded as a file identified by a File Entry ICB (VAT ICB) that allows great flexibility in building the table. The VAT ICB is the last sector recorded in any transaction. The VAT itself may be recorded at any location.

The VAT shall be identified by a File Entry ICB with a file type of 248. This ICB shall be the last valid data sector recorded. Error recovery schemes can find the last valid VAT by finding ICBs with file type 248.

This file, when small, can be embedded in the ICB that describes it. If it is larger, it can be recorded in a sector or sectors preceding the ICB. The sectors do not have to be contiguous, which allows writing only new parts of the table if desired. This allows small incremental updates, even on disks with many directories.

When the VAT is small (a small number of directories on the disk), the VAT is updated by writing a new file ICB with the VAT embedded. When the VAT becomes too large to fit in the ICB, writing a single sector with the VAT and a second sector with the ICB is required. Beyond this point, more than one sector is required for the VAT. However, as multiple extents are supported, updating the VAT may consist of writing only the sector or sectors that need updating and writing the ICB with pointers to all of the pieces of the VAT.

The Virtual Allocation Table is used to redirect requests for certain information to the proper logical location. The indirection provided by this table provides the appearance of direct overwrite capability. For example, the ICB describing the root directory could be referenced as virtual sector 1. A virtual sector is contained in a partition identified by a Virtual Partition Map entry. Over the course of updating the disk, the root directory may change. When it changes, a new sector describing the root directory is written, and its Logical Block Address is recorded as the Logical Block Address corresponding to virtual sector 1. Nothing that references virtual sector 1 needs to change, as it still points to the most current virtual sector 1 that exists, even though it exists at a new Logical Block Address.

The use of virtual addressing allows any desired structure to become effectively Rewritable. The structure is Rewritable when every pointer that references it does so only by its Virtual Address. When a replacement structure is written, the virtual reference does not need to change. The proper entry in the VAT is changed to reflect the new Logical Block Address of the corresponding Virtual Address and all virtual references then indirectly point to the new structure. All structures that require updating, such as directory ICBs, shall be referenced by a Virtual Address. As each structure is updated, its corresponding entry in the VAT ICB shall be updated.

The VAT shall be recorded as a sequence of Uint32 entries in a file. Each entry shall be the offset, in sectors, into the physical partition in which the VAT is located. The first entry shall be for the virtual partition sector 0, the second entry for virtual partition sector 1, etc. The Uint32 entries shall follow the VAT header. The entry for the previous VAT ICB allows for viewing the file system as it appeared in an earlier state. If this field is #FFFFFFFF, then no such ICB is specified.

Virtual Allocation Table structure

Offset	Length	Name	Contents
0	2	Length of Header (=L_HD)	Uint16
2	2	Length of Implementation Use (=L_IU)	Uint16
4	128	Logical Volume Identifier	dstring
132	4	Previous VAT ICB location	Uint32
136	4	Number of Files	Uint32
140	4	Number of Directories	Uint32
144	2	Minimum UDF Read Revision	Uint16
146	2	Minimum UDF Write Revision	Uint16
148	2	Maximum UDF Write Revision	Uint16
150	2	Reserved	#00 bytes
152	L_IU	Implementation Use	bytes
152 + L_IU	4	VAT entry 0	Uint32
156 + L_IU	4	VAT entry 1	Uint32
…	…	…	…
Information Length - 4	4	VAT entry n	Uint32

Length of Header - Indicates the amount of data preceding the VAT entries. This value shall be 152 + L_IU.

Length of Implementation Use - Shall specify the number of bytes in the Implementation Use field. If this field is non-zero, the value shall be at least 32 and be an integral multiple of 4.

Logical Volume Identifier - Shall identify the logical volume. This field shall be used by implementations instead of the corresponding field in the Logical Volume Descriptor. The value of this field should be the same as the field in the LVD until changed by the user.

Previous VAT ICB Location - Shall specify the logical block number of an earlier VAT ICB in the partition identified by the Partition Map entry. If this field is #FFFFFFFF, no such ICB is specified.

Number of Files - Defined in 2.2.6.4. The contents of this field shall be used instead of the corresponding LVID field.

Number of Directories - Defined in 2.2.6.4. The contents of this field shall be used instead of the corresponding LVID field.

Minimum UDF Read Revision - Defined in 2.2.6.4. The contents of this field shall be used instead of the corresponding LVID field.

Minimum UDF Write Revision - Defined in 2.2.6.4. The contents of this field shall be used instead of the corresponding LVID field.

Maximum UDF Write Revision - Defined in 2.2.6.4. The contents of this field shall be used instead of the corresponding LVID field.

Implementation Use - If non-zero in length, shall begin with an EntityID identifying the usage of the remainder of the Implementation Use area.

VAT Entry - VAT entry *n* shall identify the logical block number of the virtual block *n*. An entry of #FFFFFFFF indicates that the virtual sector is currently unused. The LBN specified is located in the partition identified by the Partition Map entry. The number of entries in the table can be determined from the VAT file size in the ICB:

Number of entries = (Information Length - L_HD) / 4.

2.2.12 Sparing Table

Certain disk/drive systems do not perform defect management (eg. CD-RW). A Sparing Table is used to provide an apparent defect-free space for these systems. Certain media can only be written in groups of sectors ("packets"), further complicating relocation: a whole packet must be relocated rather than only the sectors being written. To address this issue a sparable partition is identified in the Partition Map, which further identifies the location of the Sparing Tables. The Sparing Table identifies relocated areas on the media. Sparing Tables are identified by a Sparable Partition Map. Sparing Tables shall not be recorded on disk/drive systems that perform defect management.

Sparing Tables point to space allocated for sparing and contains a list of mappings of defective sectors to their replacements. Separate copies of the Sparing Tables shall be recorded in separate packets. All instances of the Sparing Table shall be kept up to date.

Partitions map logical space to physical space. Normally, this is a linear mapping where an offset and a length are specified. A sparable partition is based on this mapping, where the offset and length of a partition within physical space is specified by a Partition Descriptor (see 2.2.14). A sparable partition shall begin and end on a packet boundary. The Sparing Table further specifies an exception list of logical to physical mappings. All mappings are one fixed packet or ECC block in length. The Packet Length (in blocks) is specified in the Sparable Partition Map.

Available sparing areas and instances of the Sparing Table may be anywhere on the media, either inside or outside of a partition. If overlapping with a partition, the overlapping part shall be marked as allocated and shall be included in the Non-Allocatable Space Stream. The mapped locations should be filled in at format time; the original locations are assigned dynamically as errors occur. Each Sparing Table shall be structured as shown below.

Sparing Table layout

BP	Length	Name	Contents
0	16	Descriptor Tag	tag = 0
16	32	Sparing Identifier	EntityID
48	2	Reallocation Table Length (=RT_L)	Uint16
50	2	Reserved	#00 bytes
52	4	Sequence Number	Uint32
56	8*RT_L	Map Entry	Map Entries

This structure may be larger than a single sector if necessary.

- Descriptor Tag
  Contains a Tag Identifier of 0, which indicates that the format of the Descriptor Tag is not specified by ECMA 167. All other fields of the Descriptor Tag shall be valid, as if the Tag Identifier were one of the values defined by ECMA 167.
- Sparing Identifier:
  - Flags = 0
  - Identifier = `*UDF Sparing Table`
  - Identifier Suffix is recorded as defined in 2.1.5
- Reallocation Table Length
  Indicates the number of entries in the Map Entry table.
- Sequence Number
  Contains a number that shall be incremented each time the Sparing Table is updated.
- Map Entry
  A map entry is described in the table below. Maps shall be sorted in ascending order by the Original Location field.

Map Entry description

RBP	Length	Name	Contents
0	4	Original Location	Uint32
4	4	Mapped Location	Uint32

- Original Location
  Logical Block Address of the packet to be spared. The address of a packet is the address of the first user data block of a packet. If this field is #FFFFFFFF, then this entry is available for sparing. If this field is #FFFFFFF0, then the corresponding mapped location is marked as defective and should not be used for mapping. Original Locations of #FFFFFFF1 through #FFFFFFFE are reserved.
- Mapped Location
  Physical Block Address of active data. Requests to the original packet location are redirected to the packet location identified here. All Mapped Location entries shall be valid, including those entries for which the Original Location is #FFFFFFF0, #FFFFFFFF, or reserved. If the mapped location overlaps a partition, that partition shall have that space marked as allocated and that space shall be part of the Non-Allocatable Space Stream.

2.2.13 Metadata Partition

The files and policies defined in this section facilitate rapid location of all metadata in the volume, promote clustering of ICBs / directory information, and optionally facilitate duplication of all metadata. This will, in most cases, greatly speed file system repair operations by eliminating the need to perform an exhaustive media scan, or directory traversal, solely for the purpose of locating ICBs. The clustering of metadata will also significantly improve performance of metadata intensive implementation operations. When the metadata duplication option is chosen, file system robustness to media damage is increased, at some cost to performance.

When a Type 2 Metadata Partition Map is recorded, the Metadata File, Metadata Mirror File and Metadata Bitmap File shall also be recorded and maintained. The exception is that a Metadata Bitmap File shall not be recorded for a read-only partition and for a pseudo-overwritable partition.

The allocation descriptors of the Metadata Mirror File File Entry shall either:

- reference the same extents in the physical/sparable partition as referenced by the allocation descriptors of the Metadata File - in this case the *Duplicate Metadata Flag* in the Metadata Partition Map *Flags* field shall not be set.

OR

- reference different extents thus duplicating all metadata.- in this case the *Duplicate Metadata Flag* in the Metadata Partition Map *Flags* field shall be set.

The File Entries for the Metadata, Metadata Mirror and Metadata Bitmap files shall not be referenced by any structure other than the Metadata Partition Map and shall have a

File Link Count of 0. These files, when present, shall be recorded in the physical/sparable partition referenced by the Metadata Partition Map.

The Metadata Partition Map (see 2.2.10) defines a partition space in which all metadata (FSD, ICBs, Allocation Descriptors, and directory data) shall be recorded, with the sole exception of the ICBs and data comprising the Metadata, Metadata Mirror, and Metadata Bitmap files as described above.

File Entries describing directories or Stream Directories shall use either “immediate” allocation (i.e. the data is embedded in the File Entry - see ECMA 167 4/14.6.8 flag bits 0-2) or SHORT_ADs to describe the data space of the directory, since this data resides in the Metadata Partition along with the File Entry itself.

File Entries describing any other type of file data (including Named Streams) shall use either “immediate” allocation, or LONG_ADs that shall reference the physical or sparable partition referenced by the Metadata Partition Map, to describe the data space of the file. In the special two partitions case mentioned in 2.2.10, with a read-only partition and an overwritable partition on one volume, the data space of the file or Named Stream may also be located in the read-only partition.

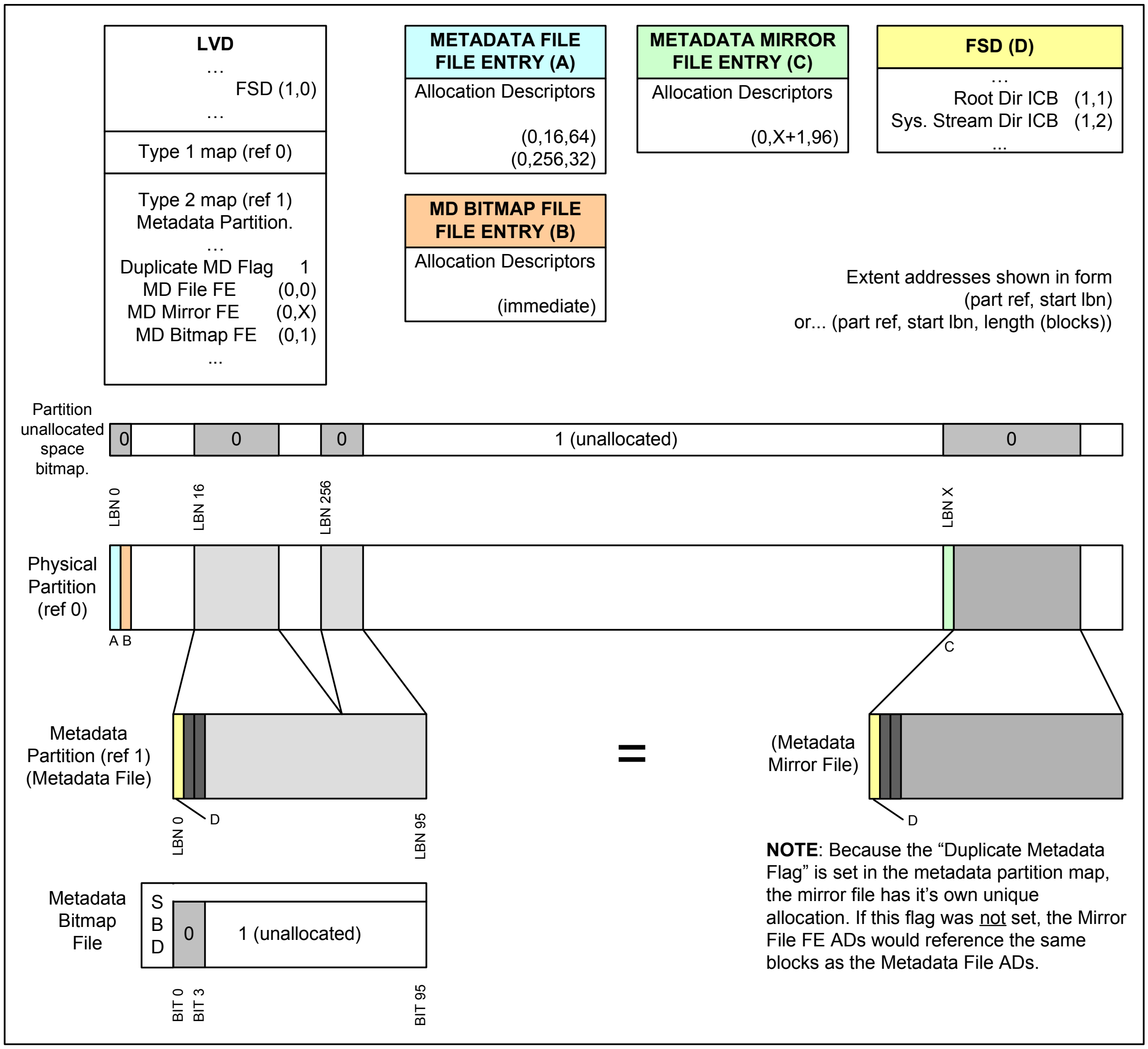
The *Extent Location* field of any Allocation Descriptor referencing data recorded in the Metadata Partition shall be interpreted as a block offset into the Metadata File. For example logical block 40 in the Metadata Partition corresponds to byte offset (40 * logical block size) in the Metadata File, which in turn (through the Allocation Descriptors for the Metadata File) corresponds to some logical block in the associated physical/sparable partition.

Implementations shall support both the duplicate and shared allocation modes for the Metadata Mirror File (see above and 2.2.10, Metadata Partition Map, *Flags* field). The File Entry for the Metadata Mirror shall be actively maintained along with the Metadata File File Entry, but should be updated after the Metadata File File Entry.

If the *Duplicate Metadata Flag* is set in the Metadata Partition Map *Flags* field, the Metadata Mirror File shall be maintained dynamically so that it contains identical contents to the Metadata File at all times. Unused logical blocks in the Metadata File and Metadata Mirror File may not be identical. In this case blocks in the Metadata Partition may be read from the same offset in either the Metadata Mirror File or the Metadata File. Data should be written first to the Metadata File and second to the Metadata Mirror File.

When the *Duplicate Metadata Flag* in the Metadata Partition Map *Flags* field is set, implementations and repair utilities should consider the Metadata File content to be primary over that of the Metadata Mirror File. For example, a repair utility could repair the volume based on metadata read from the Metadata File (excepting unreadable portions which would be read from the Mirror) and then replace the contents of the Metadata Mirror File with that of the (now consistent) Metadata File.

Logical blocks allocated to the Metadata or Metadata Mirror File shall be marked as allocated in the partition Unallocated Space Bitmap, therefore a mechanism to determine available blocks within the Metadata Partition is needed. This is accomplished through the Metadata Bitmap File. A Metadata Bitmap File shall not be recorded for a read-only partition and for a pseudo-overwritable partition.



NOTE: The LBN values used in the diagram above are for illustrative purposes only and are not fixed. The partition reference numbers used are determined by the order of the Partition Maps in the LVD.

A more detailed description of these files and how they are used follows in section 2.2.13.1.

2.2.13.1 Metadata File (and Metadata Mirror File)

These files shall have the values of 250 (main) and 251 (Mirror) recorded in the ICB Tag *File Type* fields of their File Entries. The *UniqueID* field of these File Entries shall have a value of zero.

The Allocation Descriptors (see 2.3.10) of these files shall at all times:

- Be SHORT_ADs (referencing space in the same physical/sparable partition in which the ICB resides).
- Not specify an extent of type "not recorded but allocated".
- Extents of type "recorded and allocated" or "not allocated" shall have an extent length that is an integer multiple of (*Allocation Unit Size* multiplied by logical block size). The *Allocation Unit Size* is specified in the Metadata Partition Map.
- Extents of type "recorded and allocated" shall have a starting logical block number that is an integer multiple of the *Alignment Unit Size* specified in the Metadata Partition Map.

The *Information Length* field of the File Entries for these files shall be equal to ((number of blocks described by the ADs) multiplied by logical block size).

The Allocation Descriptors for this file shall describe only logical blocks which contain one of the below data types. No user data or other metadata may be referenced.

- FSD
- Terminating Descriptor
- ICB
- Extent of Allocation Descriptors (see 2.3.11)
- Directory or Stream Directory data (i.e. FIDs)
- An unused block that is available for use

NOTE: The File Entry and possible Allocation Extent Descriptors of the Metadata File should be recorded as far apart (physically) as possible from those of the Metadata Mirror File. The same counts for the allocated extents of these two files in the case that the *Duplicate Metadata Flag* in the Metadata Partition Map is set. Typically, recording far apart is achieved by maximizing the difference between the start LBNs of the descriptors and extents belonging to the file and its mirror. Some drive/media combinations support "background physical formatting" (see 6.13 and 6.14) or "incremental formatting", and implementations using such features should consider this when locating the Metadata Files and data. In such cases it may be practically impossible to position the files far apart without impacting the early eject time / media readability.

The *Access Time* and *Modification Time* fields of the Metadata File and Mirror File File Entries shall be set to legal values at format time but need not be updated by a file system.

The File Entries for the Metadata File and Metadata Mirror File shall have NULL Stream Directory ICB and Extended Attribute ICB fields.

2.2.13.2 Metadata Bitmap File

This file shall have a value of 252 recorded in the ICB Tag *File Type* field of its File Entry. The *UniqueID* field of this File Entry shall have a value of zero.

This file contains a Space Bitmap Descriptor describing the utilization of blocks allocated to the Metadata File (i.e. this is a bitmap describing allocated space for the Metadata Partition). Bit zero of the bitmap corresponds to the first block in the aforementioned file, bit one to the second, and so on. This also applies to the Metadata Mirror File since contents of the two files are identical (regardless of the *Duplicate Metadata Flag* in the Metadata Partition Map *Flags* field).

If a bit in this bitmap is set (one) then the corresponding blocks within the Metadata File and Metadata Mirror File are available for use by new metadata.

NOTE: When the *Duplicate Metadata Flag* in the Metadata Partition Map *Flags* field is not set, these blocks are one and the same, since the Allocation Descriptors for the Metadata Mirror File reference the same blocks as those of the Metadata File.

If a bit in this bitmap is clear (zero) then the corresponding blocks are not available for use – i.e. they are either in use, or fall within an unallocated region of the Metadata File.

Other requirements for the Metadata Bitmap File:

- The descriptor tag *DescriptorCRCLength* field for this SBD shall be set to zero or 8. The value of 8 is recommended.
- The Allocation Descriptors for the Metadata Bitmap File shall not include any Allocation Descriptors of type “not allocated”.
- The *Information Length* field of the File Entry for this file shall equal the size of the SBD (NOTE: SBD size includes the bitmap portion).
- There shall be one bit in the bitmap for every block in the Metadata Partition.
- The *Access Time* and *Modification Time* fields of the Metadata Bitmap File File Entry shall be set to legal values at format time but need not be updated by a file system.
- The Metadata Bitmap File File Entry shall have NULL *Stream Directory ICB* (if extended FE) and *Extended Attribute ICB* fields.
- The descriptor *Tag Location* field of this SBD shall be set to the logical block number of the first block allocated to the Metadata Bitmap File.

2.2.13.3 Procedure for allocating blocks for new metadata.

Search for a set (one) bit in the Metadata Bitmap File, and clear it. The corresponding block within the Metadata Partition (Metadata and Metadata Mirror (if duplicate mode) files) may then be used for the new data. If there are no set (one) bits then the Metadata File (and Mirror if duplicate) must be extended as described in section 2.2.13.5 below.

2.2.13.4 Procedure for de-allocating metadata blocks.

Set (to one) the bit(s) in the Metadata Bitmap File corresponding to the block number(s) of the data within the Metadata Partition that is being de-allocated.

2.2.13.5 Recommended procedure for extending the Metadata Partition

These changes should be written to the device before the new blocks are allocated for use by metadata. It would be undesirable for such changes to sit in an implementation's write cache for so long that new metadata assigned to the blocks being described by the changes was written to the media first.

1. Verify that there is enough space in the Metadata File and Metadata Mirror File Allocation Descriptor chains for a new Allocation Descriptor. If not then allocate a new Allocation Descriptor extent.
2. Verify that the Metadata Bitmap File file allocation is large enough to extend the bitmap to describe the additional blocks added to the Metadata File, and if not then allocate block(s) for the Metadata Bitmap file.
3. Allocate a new extent of blocks (for the Metadata File) observing the size and alignment requirements specified in 2.2.13.1.
4. If the *Duplicate Metadata Flag* in the Metadata Partition Map Flags field is set, allocate a second extent of blocks observing the size and alignment requirements specified in 2.2.13.1, ideally as far away as possible from the first allocation (for the Metadata Mirror File).
5. Add a new Allocation Descriptor to the Metadata File, or modify existing descriptors, to reference the first newly allocated extent. If the *Duplicate Metadata Flag* in the Metadata Partition Map *Flags* field is <u>not</u> set, modify the Metadata Mirror File ADs to reference the same extent.
6. If a second extent of blocks was allocated above, add to the Metadata Mirror File a new Allocation Descriptor, or modify existing ADs, to reference this second extent.
7. If the new extents were added at the end of the Metadata File then increase the FE *Information Length* for the Metadata File, and Mirror, to include the new blocks.
8. If the Metadata Bitmap File was extended, increase its FE *Information Length* field to include the bits describing the additional blocks allocated to the Metadata Files.
9. Set (set to one) the bits in the Metadata Bitmap File which correspond to the extent just added to the Metadata File, to indicate the blocks are available for use by new metadata.

2.2.13.6 Recommended procedure for reclaiming space from the Metadata Partition

Blocks allocated to the Metadata File, and its mirror, shall only be returned to the volume in one of the following two ways:

- Truncation of the Metadata File and its mirror.
- Marking the AD(s) for a region of the Metadata File, and it's mirror, as sparse (not allocated) and setting the corresponding bits in the Metadata Bitmap File to zero, indicating these blocks are not available for use.

Any region to be removed shall:

- Currently contain no referenced metadata (i.e. all corresponding bits in the Metadata Bitmap File shall already be set (one)).
- Match the size/alignment restrictions laid down in section 2.2.13.1.

In the truncation case (Metadata Partition being truncated):

1. Update the SBD in the Metadata Bitmap File to reduce the bitmap size.
2. Update the Metadata Bitmap File File Entry *Information Length* to reflect the decreased bitmap size.
3. Update the Metadata File, and mirror, File Entry *Information Length* fields to 'remove' the region.
4. Mark the de-allocated blocks as available in the partition Unallocated Space Bitmap.

In the mark sparse case (region in middle of Metadata Partition being removed):

1. Clear the corresponding bits in the Metadata Bitmap File to zero.
2. Generate sparse (not allocated) Allocation Descriptor(s) in the Metadata File (and its mirror) for the region being de-allocated.
3. Mark the de-allocated blocks as available in the partition Unallocated Space Bitmap.

2.2.14 Partition Descriptor

```
struct PartitionDescriptor {                    /* ECMA 167 3/10.5 */
        struct tag          DescriptorTag;
        Uint32              VolumeDescriptorSequenceNumber;
        Uint16              PartitionFlags;
        Uint16              PartitionNumber;
        struct EntityID     PartitionContents;
        byte                PartitionContentsUse[128];
        Uint32              AccessType;
        Uint32              PartitionStartingLocation;
        Uint32              PartitionLength;
        struct EntityID     ImplementationIdentifier;
        byte                ImplementationUse[128];
        byte                Reserved[156];
}
```

2.2.14.1 Struct EntityID PartitionContents

For more information on the proper handling of this field see section 2.1.5 on *Entity Identifier*.

2.2.14.2 Uint32 AccessType

Besides the values for Access Type as defined in ECMA 167 3/10.5.7, UDF defines that the value zero shall be used for an Access Type named pseudo-overwritable. This Access Type value shall be used for partitions that support the Pseudo OverWrite Method as described in appendix 6.15.

A partition with Access Type 3 (rewritable) *shall* define a Freed Space Bitmap or a Freed Space Table, see 2.3.3. All other partitions *shall not* define a Freed Space Bitmap or a Freed Space Table.

For some Rewritable/Overwritable media types there may be confusion between partition Access Types 3 (rewritable) and 4 (overwritable).

Rewritable partitions are used on media that require some form of preprocessing before re-writing data (for example legacy MO). Such partitions shall use Access Type 3.

Overwritable partitions are used on media that *do not* require preprocessing before overwriting data (for example: CD-RW, DVD-RW, DVD+RW, DVD-RAM, BD-RE, HD DVD-Rewritable). Such partitions shall use Access Type 4.

If the value of Access Type is not equal to any of the defined Access Type values or if the combination of the medium and drive is not capable of performing the write action denoted by the Access Type value, the partition shall be handled as a read-only partition (e.g. an overwritable partition on a Write-Once medium or in a Read-Only drive).

NOTE: The above rule is important in order to enable read-only access by a UDF 2.50 implementation for media with a higher UDF revision (e.g. UDF 2.60) using a pseudo-overwritable partition and a Minimum UDF Read Revision value of 2.50.

2.2.14.3 Uint32 PartitionStartingLocation

For a Sparable Partition, the value of this field shall be an integral multiple of the Packet Length. The Packet Length is defined in the Sparable Partition Map.

For a physical partition, the value of this field shall be an integral multiple of ("ECC Block Size" (divided by) sector size) for the media (See 1.3.2 for definition of ECC Block Size).

2.2.14.4 Uint32 PartitionLength

For a Sparable Partition, the value of this field shall be an integral multiple of the Packet Length. The Packet Length is defined in the Sparable Partition Map.

2.2.14.5 Struct EntityID ImplementationIdentifier

For more information on the proper handling of this field see section 2.1.5 on *Entity Identifier*.

2.2.14.6 byte PartitionContentsUse[128]

The Partition Contents Use field contains the Partition Header Descriptor as defined in 2.3.3.

2.3 Part 4 - File Structure

2.3.1 Descriptor Tag

```
struct tag {                /* ECMA 167 4/7.2 */
        Uint16         TagIdentifier;
        Uint16         DescriptorVersion;
        Uint8          TagChecksum;
        byte           Reserved;
        Uint16         TagSerialNumber;
        Uint16         DescriptorCRC;
        Uint16         DescriptorCRCLength;
        Uint32         TagLocation;
}
```

NOTE: The value zero for TagIdentifier is not defined by ECMA 167, but it is used by UDF for the Sparing Table.

2.3.1.1 Uint16 TagSerialNumber

👓 Ignored. Intended for disaster recovery.

✍ Shall be set to the *Tag Serial Number* value for the Anchor Volume Descriptor Pointers on this volume.

The same applies as for volume structure *Tag Serial Number* values, see 2.2.1.1 and 2.1.6.

2.3.1.2 Uint16 DescriptorCRCLength

The same applies as for volume structure *DescriptorCRCLength* values, see 2.2.1.2.

2.3.1.3 Uint32 TagLocation

For structures referenced via a virtual address (i.e. referenced through the VAT), this value shall be the virtual address, not the physical or logical address.

2.3.2 File Set Descriptor

```
struct FileSetDescriptor { /* ECMA 167 4/14.1 */
        struct tag          DescriptorTag;
        struct timestamp    RecordingDateandTime;
        Uint16              InterchangeLevel;
        Uint16              MaximumInterchangeLevel;
        Uint32              CharacterSetList;
        Uint32              MaximumCharacterSetList;
        Uint32              FileSetNumber;
        Uint32              FileSetDescriptorNumber;
        struct charspec     LogicalVolumeIdentifierCharacterSet;
        dstring             LogicalVolumeIdentifier[128];
        struct charspec     FileSetCharacterSet;
        dstring             FileSetIdentifier[32];
        dstring             CopyrightFileIdentifier[32];
        dstring             AbstractFileIdentifier[32];
        struct long_ad      RootDirectoryICB;
        struct EntityID     DomainIdentifier;
        struct long_ad      NextExtent;
        struct long_ad      SystemStreamDirectoryICB;
        byte                Reserved[32];
}
```

Only one *File Set Descriptor* shall be recorded. On WORM media, multiple *File Sets* may be recorded.

The UDF provision for multiple File Sets is as follows:

- Multiple *File Set*s are only allowed on WORM media.
- The default *File Set* shall be the one with the highest *FileSetNumber*.
- Only the default *File Set* may be flagged as writable. All other *File Sets* in the sequence shall be flagged *HardWriteProtect* (see 2.1.5.3).
- No writable *File Set* shall reference any metadata structures which are referenced (directly or indirectly) by any other *File Set*. Writable *File Set*s may, however, reference the actual file data extents.

Within a *File Set* on WORM, if all files and directories have been recorded with ICB Strategy Type 4, then the *Domain Identifier* of the corresponding *File Set Descriptor* shall be marked as *HardWriteProtected*.

The intended purpose of multiple *File Set*s on WORM is to support the ability to have multiple archive images on the media. For example one *File Set* could represent a backup of a certain set of information made at a specific point in time. The next *File Set* could represent another backup of the same set of information made at a later point in time.

2.3.2.1 Uint16 InterchangeLevel

☞ Interpreted as specifying the current interchange level (as specified in ECMA 167 4/15), of the contents of the associated file set and the restrictions implied by the specified level.

✍ Shall be set to a level of 3.

An implementation shall enforce the restrictions associated with the specified current *Interchange Level*.

2.3.2.2 Uint16 MaximumInterchangeLevel

☞ Interpreted as specifying the maximum interchange level of the contents of the associated file set. This value restricts to what the current *Interchange Level* field may be set.

✍ Shall be set to level 3.

2.3.2.3 Uint32 CharacterSetList

☞ Interpreted as specifying the character set(s) specified by any field, whose contents are specified to be a charspec, of any descriptor specified in Part 4 of ECMA 167 and recorded in the file set described by this descriptor.

✍ Shall be set to indicate support for CS0 only as defined in 2.1.2.

2.3.2.4 Uint32 MaximumCharacterSetList

☞ Interpreted as specifying the maximum supported character set in the associated file set and the restrictions implied by the specified level.

✍ Shall be set to indicate support for CS0 only as defined in 2.1.2.

2.3.2.5 struct charspec LogicalVolumeIdentifierCharacterSet

☞ Interpreted as specifying the d-characters allowed in the *Logical Volume Identifier* field.

✍ Shall be set to indicate support for CS0 as defined in 2.1.2.

2.3.2.6 struct charspec FileSetCharacterSet

☞ Interpreted as specifying the d-characters allowed in dstring fields defined in Part 4 of ECMA 167 that are within the scope of the File Set Descriptor.

✍ Shall be set to indicate support for CS0 as defined in 2.1.2.

2.3.2.7 struct EntityID DomainIdentifier

☞ Interpreted as specifying a domain specifying rules on the use of, and restrictions on, certain fields in the descriptors. If this field is NULL then it is ignored, otherwise the *Entity Identifier* rules are followed.

✍ This field shall indicate that the scope of this *File Set Descriptor* conforms to the domain defined in this document, therefore the *Domain Identifier* ID value shall be set to:

"*OSTA UDF Compliant"

As described in section 2.1.5 on *Entity Identifier* the *Identifier Suffix* field of this *EntityID* shall contain the revision of this document for which the contents of the Logical Volume is compatible. For more information on the proper handling of this field see section 2.1.5.

NOTE: The *Identifier Suffix* field of this EntityID contains *SoftWriteProtect* and *HardWriteProtect* flags.

2.3.3 Partition Header Descriptor

```
struct PartitionHeaderDescriptor {                /* ECMA 167 4/14.3 */
        struct short_ad    UnallocatedSpaceTable;
        struct short_ad    UnallocatedSpaceBitmap;
        struct short_ad    PartitionIntegrityTable;
        struct short_ad    FreedSpaceTable;
        struct short_ad    FreedSpaceBitmap;
        byte               Reserved[88];
}
```

The Partition Header Descriptor is recorded in the Partition Contents Use field of the Partition Descriptor.
As a point of clarification the logical blocks represented as *Unallocated* are blocks that are ready to be written without any preprocessing. In the case of Rewritable media this would be a write without an erase pass. The logical blocks represented as *Freed* are blocks that are not ready to be written, and require some form of preprocessing. In the case of Rewritable media this would be a write with an erase pass. See section 2.2.14.2 for further detail regarding media classification.

NOTE 1: The use of Space Tables or Space Bitmaps shall be consistent across a Logical Volume. Space Tables and Space Bitmaps shall not both be used at the same time within a Logical Volume.

NOTE 2: A Space Table or Space Bitmap shall not be recorded for a read-only partition, a pseudo-overwritable partition or for a file system using a VAT.

2.3.3.1 struct short_ad PartitionIntegrityTable

Shall be set to all zeros since PartitionIntegrityEntrys are not used.

2.3.4 File Identifier Descriptor

```
struct FileIdentifierDescriptor {                    /* ECMA 167 4/14.4 */
        struct tag          DescriptorTag;
        Uint16              FileVersionNumber;
        Uint8               FileCharacteristics;
        Uint8               LengthofFileIdentifier;
        struct long_ad      ICB;
        Uint16              LengthOfImplementationUse;
        byte                ImplementationUse[];
        char                FileIdentifier[];
        byte                Padding[];
}
```

The *File Identifier Descriptor* shall be restricted to the length of at most one Logical Block.

NOTE 1: All UDF directories shall include a File Identifier Descriptor that indicates the location of the parent directory. The File Identifier Descriptor describing the parent directory shall be the first File Identifier Descriptor recorded in the directory. The parent directory of the Root Directory shall be the Root Directory, as stated in ECMA 167 4/8.6

NOTE 2: On logical volumes where a Metadata Partition Map is recorded, all directory and stream directory data shall be recorded in the Metadata Partition (see 2.2.10), however the data space of Named Streams shall be recorded in physical space.

2.3.4.1 Uint16 FileVersionNumber

☞ There shall be only one version of a file as specified below with the value being set to 1.

✍ Shall be set to 1.

2.3.4.2 Uint8 FileCharacteristics

2.3.4.2.1 Deleted bit

The Deleted bit may be used to mark a file or directory as deleted instead of removing the FID from the directory, which requires rewriting the directory from that point to the end. If the space for the file or directory is deallocated, the implementation shall set the ICB field to zero, as all fields in a FID must be valid even if the Deleted bit is set. See ECMA 167 4/14.4.3 note 21 and 4/14.4.5.

ECMA 167 4/8.6 requires that the File Identifiers (and File Version Numbers, which shall always be 1) of all FIDs in a directory shall be unique. While the standard is silent on whether FIDs with the Deleted bit set are subject to this requirement, the intent is that they are not. FIDs with the Deleted bit set are not subject to the uniqueness requirement, as interpreted by UDF

In order to assist a UDF implementation that may have read the standard without this interpretation, implementations shall follow these rules when a FID's Deleted bit is set:

If the compression ID of the File Identifier is 8, rewrite the compression ID to 254. If the compression ID of the File Identifier is 16, rewrite the compression ID to 255. Leave the remaining bytes of the File Identifier unchanged

In this way a utility wishing to undelete a file or directory can recover the original name by reversing the rewrite of the compression ID.

NOTE: Implementations should re-use FIDs that have the Deleted bit set to one and ICBs set to zero in order to avoid growing the size of the directory unnecessarily.

2.3.4.2.2 Parent bit and Directory bit

There is a flaw in the following statement in ECMA 167 4/14.4.3, below figure 13:

> "If the Parent bit is set to ONE, then the Directory bit shall be set to ONE."

In spite of this statement, the Directory bit in a parent FID shall only be set to ONE if the FID identifies a directory or the System Stream Directory. If the parent FID identifies a file, the Directory bit shall be set to ZERO. The latter is the case for a parent FID in a Stream Directory that is attached to a file.

2.3.4.3 struct long_ad ICB

The *Implementation Use* bytes of the long_ad in all *File Identifier Descriptors* shall be used to store the UDF UniqueID for the file and directory namespace.

The *Implementation Use* bytes of a long_ad hold an ADImpUse structure as defined by 2.3.10.1. The four impUse bytes of that structure will be interpreted as a Uint32 holding the UDF UniqueID.

ADImpUse structure holding UDF UniqueID

RBP	Length	Name	Contents
0	2	Flags *(see 2.3.10.1)*	Uint16
2	4	UDF UniqueID	Uint32

Section 3.2.1 Logical Volume Header Descriptor describes how *UDF UniqueID* field in Implementation Use bytes of the long_ad in the File Identifier Descriptor and the *UniqueID* field in the File Entry and Extended File Entry are set.

2.3.4.4 Uint16 LengthofImplementationUse

- ☞ Shall specify the length of the *ImplementationUse* field.

- ✍ Shall specify the length of the *ImplementationUse* field. This field may contain zero, indicating that the ImplementationUse field has not been used. Otherwise, this field shall contain at least 32 as required by 2.3.4.5.

When writing a File Identifier Descriptor to Write-Once media, to ensure that the Descriptor Tag field of the next FID will never span a block boundary, if there are less than 16 bytes remaining in the current block after the FID, the length of the FID shall be increased (using the Implementation Use field) enough to prevent this. Remember that in the latter case, the Implementation Use field shall be at least 32 bytes.

2.3.4.5 byte ImplementationUse[]

- ☞ If the *LengthofImplementationUse* field is non ZERO then the first 32 bytes of this field shall be interpreted as specifying the implementation identifier *EntityID* of the implementation which last modified the *File Identifier Descriptor*.

- ✍ If the *LengthofImplementationUse* field is non ZERO then the first 32 bytes of this field shall be set to the implementation identifier *EntityID* of the current implementation.

NOTE: For additional information on the proper handling of this field refer to section 2.1.5 on *Entity Identifier*.

This field allows an implementation to identify which implementation last created and/or modified a specific *File Identifier Descriptor*.

2.3.4.6 char FileIdentifier[]

Contains a File Identifier stored in the OSTA Compressed Unicode format, see 2.1.1. The byte length of this field shall be greater than 1 with the sole exception of 0 for a parent FID. If the Deleted bit is set in the File Characteristics field of this File Identifier Descriptor, then see 2.3.4.2.1 for additional rules. If the Deleted bit is not set, then the Unicode representation of the File Identifier shall be unique in this directory. This requires not only byte-wise uniqueness as required by ECMA 167 4/8.6, but also uniqueness of the Unicode identifier resulting from uncompress of the OSTA Compressed Unicode format.

2.3.5 ICB Tag

```
struct icbtag {                 /* ECMA 167 4/14.6 */
        Uint32          PriorRecordedNumberofDirectEntries;
        Uint16          StrategyType;
        byte            StrategyParameter[2];
        Uint16          MaximumNumberofEntries;
        byte            Reserved;
        Uint8           FileType;
        Lb_addr         ParentICBLocation;
        Uint16          Flags;
}
```

2.3.5.1 Uint16 StrategyType

- 👓 The content of this field specifies the ICB Strategy Type used. For the purposes of read access an implementation shall support ICB Strategy Types 4 and 4096.
- ✍ Shall be set to 4 or 4096, see NOTE.

NOTE: ICB Strategy Type 4096, defined in section 6.6, is intended for use on WORM media. ICB Strategy Type 4096 is allowed only for ICBs in a partition with Access Type write-once recorded on non-sequential Write-Once media.

2.3.5.2 Uint8 FileType

As a point to clarification a value of 5 shall be used for a standard byte addressable file, not 0. The value of 248 shall be used for the VAT (refer to 2.2.11). The value of 249 shall be used to indicate a Real-Time file (see Appendix 6.11.1). File types 250, 251 and 252 shall be used for the Metadata File, Metadata Mirror File and Metadata Bitmap File respectively. See section 2.2.13 for more details. File types 253 to 255 shall not be used.

2.3.5.2.1 File Type 249

Files with File Type 249 require special commands to access the data space of this file. To avoid possible damage, if an implementation does not support these commands it shall not issue any command that would access or modify the data space of this file. This includes but is not limited to reading, writing and deleting the file.

2.3.5.3 ParentICBLocation

For ICB Strategy Type 4 this field shall not be used and contain all zero bytes. For ICB Strategy Type 4096 the use of this field is optional.

NOTE: In ECMA 167-4/14.6.7 it states, "If this field contains 0, then no such ICB is specified." This is a flaw in the ECMA 167 standard in that an implementation could store an ICB at logical block address 0. Therefore, if you decide to use this field, do not store an ICB at logical block address 0.

2.3.5.4 Uint16 Flags

Bits 0-2: These bits specify the type of allocation descriptors used. Refer to section 2.3.10 on *Allocation Descriptors* for the guidelines on choosing which type of allocation descriptor to use.

Bit 3 (*Sorted*):

☞ For OSTA UDF compliant media this bit shall indicate (ZERO) that directories may be unsorted.

✍ Shall be set to ZERO.

Bit 4 (Non-relocatable):

☞ For OSTA UDF compliant media this bit shall indicate (ONE) if the file is non-relocatable. If ONE, an implementation shall set the bit to ZERO if a modification will contravene the definition of this bit in ECMA 167 4/14.6.8.

✍ Should be set to ZERO unless required.

NOTE: This flag is **not** a lock on the file in any way. It is used to indicate that an implementation has arranged the allocation of the file to satisfy specific application requirements. In these cases, any remapping of a written block (see UDF sparable partitions) or defragmentation of the file might not be desired. If a file with this flag set to ONE is copied, then the new copy of the file should have this bit set to ZERO.

Bit 9 (*Contiguous*):

☞ For OSTA UDF compliant media this bit may indicate (ONE) that the file is contiguous. An implementation may reset this bit to ZERO to indicate that the file may be non-contiguous if the implementation can not assure that the file is contiguous.

✍ Should be set to ZERO.

Bit 11 (*Transformed*):

☞ For OSTA UDF compliant media this bit shall indicate (ZERO) that no transformation has taken place.

✍ Shall be set to ZERO.

The methods used for data compression and other forms of data transformation might be addressed in a future OSTA document.

Bit 12 (*Multi-versions*):

☞ For OSTA UDF compliant media this bit shall indicate (ZERO) that multi-versioned files are not present.

✍ Shall be set to ZERO.

2.3.6 File Entry

```
struct FileEntry {            /* ECMA 167 4/14.9 */
    struct tag          DescriptorTag;
    struct icbtag       ICBTag;
    Uint32              Uid;
    Uint32              Gid;
    Uint32              Permissions;
    Uint16              FileLinkCount;
    Uint8               RecordFormat;
    Uint8               RecordDisplayAttributes;
    Uint32              RecordLength;
    Uint64              InformationLength;
    Uint64              LogicalBlocksRecorded;
    struct timestamp    AccessTime;
    struct timestamp    ModificationTime;
    struct timestamp    AttributeTime;
    Uint32              Checkpoint;
    struct long_ad      ExtendedAttributeICB;
    struct EntityID     ImplementationIdentifier;
    Uint64              UniqueID,
    Uint32              LengthofExtendedAttributes;
    Uint32              LengthofAllocationDescriptors;
    byte                ExtendedAttributes[];
    byte                AllocationDescriptors[];
}
```

NOTE 1: The total length of a *File Entry* shall not exceed the size of one logical block.

NOTE 2: If a Metadata Partition Map is recorded in a volume then all *File Entries*, Allocation Descriptor Extents and directory data shall be recorded in the Metadata Partition – i.e. in logical blocks allocated to the Metadata and/or Metadata Mirror File (see section 2.2.13 for details including exceptions).

2.3.6.1 Uint8 RecordFormat;

- For OSTA UDF compliant media a value of zero shall indicate that the structure of the information recorded in the file is not specified by this field.
- Shall be set to ZERO.

2.3.6.2 Uint8 RecordDisplayAttributes;

- For OSTA UDF compliant media a value of zero shall indicate that the structure of the information recorded in the file is not specified by this field.
- Shall be set to ZERO.

2.3.6.3 Uint32 RecordLength;

- For OSTA UDF compliant media a value of zero shall indicate that the structure of the information recorded in the file is not specified by this field.
- Shall be set to ZERO.

2.3.6.4 Uint64 InformationLength

Only the last extent of the file body may have an extent length that is not a multiple of the block size, see ECMA 167 4/12.1 and 4/14.14.1.1.

2.3.6.5 Uint64 LogicalBlocksRecorded

For files and directories with embedded data the value of this field shall be ZERO.

2.3.6.6 struct EntityID ImplementationIdentifier;

Refer to section 2.1.5 on *Entity Identifier*.

2.3.6.7 Uint64 UniqueID

For the *root* directory of a file set this value shall be set to ZERO.

Section 3.2.1 Logical Volume Header Descriptor describes how the UDF UniqueID field in the Implementation Use bytes of the long_ad in the File Identifier Descriptor and the UniqueID field in the File Entry and Extended File Entry are set.

2.3.6.8 FileLinkCount

Hard links to a directory are not allowed. A directory File Entry shall be identified by:

- for non-root directories: exactly one FID defining the directory name
- zero or more parent FIDs if appropriate. One parent FID in each immediate child directory, if any.

For Named Stream and Stream Directory hard link restrictions, see 3.3.5.1.

2.3.7 Unallocated Space Entry

```
struct UnallocatedSpaceEntry {                  /* ECMA 167 4/14.11 */
        struct tag        DescriptorTag;
        struct icbtag     ICBTag;
        Uint32            LengthofAllocationDescriptors;
        byte              AllocationDescriptors[];
}
```

NOTE: The maximum length of an UnallocatedSpaceEntry shall be one Logical Block.

2.3.7.1 byte AllocationDescriptors[]

Only Short Allocation Descriptors shall be used.

NOTE: The upper 2 bits of the extent length field in allocation descriptors specify an extent type (ECMA 167 4/14.14.1.1). For the allocation descriptors specified for the UnallocatedSpaceEntry the type shall be set to a value of 1 to indicate *extent allocated but not recorded*, or shall be set to a value of 3 to indicate *the extent is the next extent of allocation descriptors*. This next extent of allocation descriptors shall be limited to the length of one Logical Block.

AllocationDescriptors shall be ordered sequentially in ascending location order. No overlapping *AllocationDescriptors* shall exist in the table. For example, ad.location = 2, ad.length = 2048 (logical block size = 1024) then nextad.location = 3 is not allowed. Adjacent *AllocationDescriptors* shall not be contiguous. For example ad.location = 2, ad.length = 1024 (logical block size = 1024), nextad.location = 3 is not allowed and would instead be a single *AllocationDescriptor*, ad.location = 2, ad.length = 2048. The only case where adjacent *AllocationDescriptors* may be contiguous is when the ad.length of one of the adjacent *AllocationDescriptors* is equal to the maximum *AllocationDescriptors* length.

2.3.8 Space Bitmap Descriptor

```
struct SpaceBitmap {        /* ECMA 167 4/14.12 */
        struct Tag          DescriptorTag;
        Uint32              NumberOfBits;
        Uint32              NumberOfBytes;
        byte                Bitmap[];
}
```

2.3.8.1 struct Tag DescriptorTag

There are exception rules for the SBD *DescriptorCRCLength.* If the default value for the *DescriptorCRCLength* as defined by 2.3.1.2 is not used, then *DescriptorCRCLength* shall be either zero or 8. The value of 8 is recommended.

2.3.9 Partition Integrity Entry

(See ECMA 167 4/14.13). With the functionality of the *Logical Volume Integrity Descriptor* (see section 2.2.6) this descriptor is not needed, and therefore this descriptor shall not be recorded.

2.3.10 Allocation Descriptors

When constructing the data area of a file an implementation has several types of allocation descriptors from which to choose. The following guidelines shall be followed in choosing the proper allocation descriptor to be used:

Short Allocation Descriptor - For a Logical Volume that resides on a single Volume with no intent to expand the Logical Volume beyond the single volume *Short Allocation Descriptors* should be used. For example a Logical Volume created for a standalone drive.

NOTE 1: Refer to section 2.2.2.2 on the *MaximumInterchangeLevel*.

Long Allocation Descriptor - For a Logical Volume that resides on a single Volume with intent to later expand the Logical Volume beyond the single Volume, or a Logical Volume that resides on multiple Volumes *Long Allocation Descriptors* should be used. For example a Logical Volume created for a jukebox.

NOTE 2: There is a benefit of using Long Allocation Descriptors even on a single volume, which is the support of tracking erased extents on Rewritable media. See section 2.3.10.1 for additional information.

For both Short and Long Allocation Descriptors, if the 30 least significant bits of the *ExtentLength* field is 0, then the 2 most significant bits shall be 0.

NOTE 3: For volumes in which a Virtual Partition Map is recorded:

- Allocation Descriptors identifying virtual space shall have an extent length of one block size or less. Allocation Descriptors identifying file data, directories, or stream data shall identify physical space. ICBs recorded in virtual space shall use long_ad Allocation Descriptors to identify physical space. The use of short_ad Allocation Descriptors would identify file data in virtual space if the ICB were in virtual space.

- Descriptors recorded in virtual space shall have the virtual logical block number recorded in the Tag Location field.

NOTE 4: For volumes in which a Metadata Partition Map is recorded:

- Allocation Descriptors identifying directory or stream directory data shall identify metadata space.
- Allocation Descriptors identifying file or stream data shall identify physical space.
- Allocation Descriptors recorded in metadata space shall use SHORT_ADs when identifying extents also in metadata space.
- Allocation Descriptors having an extent type of 3 (continuation) shall identify an extent in the same partition in which the type 3 descriptor itself is recorded.
- Descriptors recorded in metadata space shall have their metadata space logical block number recorded in their descriptor tag *Tag Location* field, if applicable.

2.3.10.1 Long Allocation Descriptor

```
struct long_ad {          /* ECMA 167 4/14.14.2 */
    Uint32                ExtentLength;
    Lb_addr               ExtentLocation;
    byte                  ImplementationUse[6];
}
```

To allow use of the *ImplementationUse* field by UDF and also by implementations the following structure shall be recorded within the 6-byte *Implementation Use* field.

```
struct ADImpUse
{
      Uint16 flags;
      byte   impUse[4];
}

/*
 * ADImpUse Flags (NOTE: bits 1-15 reserved for future use by UDF)
*/
#define EXTENTErased       (0x01)
```

In the interests of efficiency on *Rewritable* media that benefits from preprocessing, the EXTENTErased flag shall be set to ONE to indicate an *erased* extent. This applies only to extents of type *not recorded but allocated*.

2.3.11 Allocation Extent Descriptor

```
struct AllocationExtentDescriptor {                /* ECMA 167 4/14.5 */
        struct tag      DescriptorTag;
        Uint32          PreviousAllocationExtentLocation;
        Uint32          LengthOfAllocationDescriptors;
}
```

The Allocation Extent Descriptor does not contain the Allocation Descriptors itself. UDF will interpret ECMA 167, 4/14.5 in such a way that the Allocation Descriptors will start on the first byte following the *LengthOfAllocationDescriptors* field of the Allocation Extent Descriptor. The Allocation Extent Descriptor together with its Allocation Descriptors constitutes an extent of Allocation Descriptors. The length of an extent of Allocation Descriptors shall not exceed the logical block size. Unused bytes following the Allocation Descriptors till the end of the logical block shall have a value of #00.

2.3.11.1 Struct tag DescriptorTag

The DescriptorCRCLength of the Descriptor Tag should include the Allocation Descriptors following the Allocation Extent Descriptor. The DescriptorCRCLength shall be either 8 or 8 + LengthOfAllocationDescriptors.

2.3.11.2 Uint32 PreviousAllocationExtentLocation

👓 The previous allocation extent location shall not be used.

✍ Shall be set to 0.

2.3.12 Pathname

2.3.12.1 Path Component

```
struct PathComponent {    /* ECMA 167 4/14.16.1 */
        Uint8             ComponentType;
        Uint8             LengthofComponentIdentifier;
        Uint16            ComponentFileVersionNumber;
        char              ComponentIdentifier[ ];
}
```

2.3.12.1.1 Uint16 ComponentFileVersionNumber

- 👓 There shall be only one version of a file as specified below with the value being set to ZERO.
- ✍ Shall be set to ZERO.

2.4 Part 5 - Record Structure

Record structure files shall not be created. If they are encountered on the media and they are not supported by the implementation they shall be treated as an uninterpreted stream of bytes.

3. System Dependent Requirements

3.1 Part 1 - General

3.1.1 Timestamp

```
struct timestamp {          /* ECMA 167 1/7.3 */
        Uint16              TypeAndTimezone;
        Int16               Year;
        Uint8               Month;
        Uint8               Day;
        Uint8               Hour;
        Uint8               Minute;
        Uint8               Second;
        Uint8               Centiseconds;
        Uint8               HundredsofMicroseconds;
        Uint8               Microseconds;
}
```

3.1.1.1 Uint8 Centiseconds;

🕶 For operating systems that do not support the concept of *centiseconds* the implementation shall ignore this field.

✍ For operating systems that do not support the concept of *centiseconds* the implementation shall set this field to ZERO.

3.1.1.2 Uint8 HundredsofMicroseconds;

🕶 For operating systems that do not support the concept of *hundreds of Microseconds* the implementation shall ignore this field.

✍ For operating systems that do not support the concept of a *hundreds of Microseconds* the implementation shall set this field to ZERO.

3.1.1.3 Uint8 Microseconds;

🕶 For operating systems that do not support the concept of *microseconds* the implementation shall ignore this field.

✍ For operating systems that do not support the concept of *microseconds* the implementation shall set this field to ZERO.

3.2 Part 3 - Volume Structure

3.2.1 Logical Volume Header Descriptor

```
struct LogicalVolumeHeaderDesc {          /* ECMA 167 4/14.15 */
        Uint64          UniqueID,
        bytes           Reserved[24]
}
```

This structure is in the LVID Logical Volume Contents Use field.

3.2.1.1 Uint64 UniqueID

This field contains the *Next UniqueID* value to be used for the next new objects in the UDF UniqueID Mapping Data Stream, see 3.3.7.1. The *Next UniqueID* value is initialized to 16 because the value 0 is reserved for the root directory and System Stream Directory objects and the values 1-15 are reserved for use in Macintosh implementations. The *Next UniqueID* value monotonically increases with each assignment of a new UDF UniqueID value for a newly created object as described below. Whenever the lower 32-bits of the *Next UniqueID* value reach #FFFFFFFF, the next increment is performed by incrementing the upper 32-bits by 1, as would be expected for a 64-bit value, but the lower 32-bits "wrap" to 16 (the initialization value). After such a "wrap", the uniqueness of a 32-bits FID UDF UniqueID value can no longer be guaranteed. Therefore the UDF UniqueID Mapping Data Stream shall be removed altogether if the value of *Next UniqueID* is higher than #FFFFFFFF.

UniqueID is used whenever a new file or directory is created, or another name is linked to an existing file or directory. During a rename or move operation, the FID UniqueID value of an object shall not be changed and the values in the corresponding UDF Unique ID Mapping Entry shall remain consistent, see 3.3.7.1.2. The parent references of this mapping entry shall be updated when an object is moved to a different directory. When a FID is deleted, the mapping entry corresponding to the now unused UDF UniqueID shall not be re-used but be deleted or marked invalid. The File Identifier Descriptors and File Entries/Extended File Entries used for a Stream Directory and Named Streams associated with a file or directory do not use UniqueID; rather, the unique ID fields in these structures take their value from the UniqueID of the File Entry/Extended File Entry of the file/directory they are associated with. The same counts for File Entries/Extended File Entries used to define an Extended Attributes Space. A parent FID takes its Unique ID value from the 32 lower bits of the File Entry/Extended File Entry that is identified by the parent FID.
FIDs and File Entries of the System Stream Directory and of streams associated with the System Stream Directory shall use a UniqueID value of zero.

When a file or directory is created, this UniqueID is assigned to the UniqueID field of the File Entry/Extended File Entry, the lower 32-bits of UniqueID are assigned to UDF UniqueID in the Implementation Use bytes of the ICB field in the File Identifier Descriptor (see 2.3.4.3), and UniqueID is incremented by the policy described above.

When a name is linked to an existing file or directory, the lower 32-bits of Next UniqueID are assigned to UDF UniqueID in the Implementation Use bytes of the ICB field in the File Identifier Descriptor (see 2.3.4.3), and UniqueID is incremented by the policy described above.

The lower 32-bits shall be the same in the File Entry/Extended File Entry and its first File Identifier Descriptor, but they shall differ in subsequent FIDs.

All UDF implementations shall maintain the UDF UniqueID in the FID and UniqueID in the FE/EFE as described in this section. The LVHD in a closed Logical Volume Integrity Descriptor shall have a valid UniqueID.

For file systems using a VAT, the function of the LVHD *UniqueID* field in the LVID is taken over by the VAT ICB File Entry UniqueID field with the addition that the first UniqueID value to be used for newly created objects will be the VAT ICB UniqueID value incremented once according to the incrementing policy described for *Next UniqueID* above in this section. In this way, no other object will have the same UniqueID value as the VAT ICB File Entry.

3.3 Part 4 - File Structure

3.3.1 File Identifier Descriptor

```
struct FileIdentifierDescriptor {                /* ECMA 167 4/14.4 */
        struct tag          DescriptorTag;
        Uint16              FileVersionNumber;
        Uint8               FileCharacteristics;
        Uint8               LengthofFileIdentifier;
        struct long_ad      ICB;
        Uint16              LengthofImplementationUse;
        byte                ImplementationUse[];
        char                FileIdentifier[];
        byte                Padding[];
}
```

3.3.1.1 Uint8 FileCharacteristics

The following sections describe the usage of the *FileCharacteristics* under various operating systems.

3.3.1.1.1 MS-DOS, OS/2, Windows 95, Windows NT, Macintosh

If Bit 0 is set to ONE, the file shall be considered a "hidden" file.
If Bit 1 is set to ONE, the file shall be considered a "directory."
If Bit 2 is set to ONE, the file shall be considered "deleted."
If Bit 3 is set to ONE, the ICB field within the associated *File Identifier Descriptor* shall be considered as identifying the "parent" directory of the directory that this descriptor is recorded in.

If the file is designated as a "hidden" file, Bit 0 shall be set to ONE.
If the file is designated as a "directory," Bit 1 shall be set to ONE.
If the file is designated as "deleted," Bit 2 shall be set to ONE.

3.3.1.1.2 UNIX and OS/400

Under UNIX and OS/400 these bits shall be processed the same as specified in 3.3.1.1.1, except for hidden files which will be processed as normal non-hidden files.

3.3.2 ICB Tag

```
struct icbtag {              /* ECMA 167 4/14.6 */
        Uint32       PriorRecordedNumberofDirectEntries;
        Uint16       StrategyType;
        byte         StrategyParameter[2];
        Uint16       MaximumNumberofEntries;
        byte         Reserved;
        Uint8        FileType;
        Lb_addr      ParentICBLocation;
        Uint16       Flags;
}
```

3.3.2.1 Uint16 Flags

3.3.2.1.1 MS-DOS, OS/2, Windows 95, Windows NT

Bits 6 & 7 (*Setuid & Setgid*):

👓 Ignored.

✍ In the interests of maintaining security under environments which do support these bits; bits 6 and 7 shall be set to ZERO if any one of the following conditions are true :

- A file is created.
- The attributes/permissions associated with a file, are modified .
- A file is *written to* (the contents of the data associated with a file are modified).
- An Extended Attribute associated with the file is modified.
- A Named Stream associated with a file is modified.

Bit 8 (*Sticky*):

👓 Ignored.

✍ Shall be set to ZERO.

Bit 10 (*System*):

👓 Mapped to the MS-DOS / OS/2 system bit.

✍ Mapped from the MS-DOS / OS/2 system bit.

3.3.2.1.2 Macintosh

Bits 6 & 7 (*Setuid & Setgid*):

👓 Ignored.

✍ In the interests of maintaining security under environments, which do support these bits; bits 6 and 7 shall be set to ZERO if any one of the following conditions are true:

- A file is created.
- The attributes/permissions associated with a file, are modified.

- A file is *written to* (the contents of the data associated with a file are modified).
- An Extended Attribute associated with the file is modified.
- A Named Stream associated with a file is modified.

Bit 8 (*Sticky*):

👓 Ignored.

✍ Shall be set to ZERO.

Bit 10 (*System*):

👓 Ignored.

✍ Shall be set to ZERO.

3.3.2.1.3 UNIX

Bits 6, 7 & 8 (*Setuid, Setgid, Sticky*):

These bits are mapped to/from the corresponding standard UNIX file system bits.

Bit 10 (*System*):

👓 Ignored.

✍ Shall be set to ZERO upon file creation only, otherwise maintained.

3.3.2.1.4 OS/400

Bits 6 & 7 (*Setuid & Setgid*):

👓 Ignored.

✍ In the interests of maintaining security under environments, which do support these bits; bits 6 and 7 shall be set to ZERO if any one of the following conditions are true:

- A file is created.
- The attributes/permissions associated with a file, are modified.
- A file is *written to* (the contents of the data associated with a file are modified).
- An Extended Attribute associated with the file is modified.
- A Named Stream associated with a file is modified.

Bit 8 (*Sticky*):

👓 Ignored.

✍ Shall be set to ZERO.

Bit 10 (*System*):

👓 Ignored.

✍ Shall be set to ZERO upon file creation only, otherwise maintained.

3.3.3 File Entry

```
struct FileEntry {              /* ECMA 167 4/14.9 */
        struct tag              DescriptorTag;
        struct icbtag           ICBTag;
        Uint32                  Uid;
        Uint32                  Gid;
        Uint32                  Permissions;
        Uint16                  FileLinkCount;
        Uint8                   RecordFormat;
        Uint8                   RecordDisplayAttributes;
        Uint32                  RecordLength;
        Uint64                  InformationLength;
        Uint64                  LogicalBlocksRecorded;
        struct timestamp        AccessTime;
        struct timestamp        ModificationTime;
        struct timestamp        AttributeTime;
        Uint32                  Checkpoint;
        struct long_ad          ExtendedAttributeICB;
        struct EntityID         ImplementationIdentifier;
        Uint64                  UniqueID,
        Uint32                  LengthofExtendedAttributes;
        Uint32                  LengthofAllocationDescriptors;
        byte                    ExtendedAttributes[];
        byte                    AllocationDescriptors[];
}
```

NOTE: The total length of a *File Entry* shall not exceed the size of one logical block.

3.3.3.1 Uint32 Uid

☞ For operating systems that do not support the concept of a *user identifier* the implementation shall ignore this field. For operating systems that do support this field a value of 2^{32} - 1 shall indicate an invalid UID, otherwise the field contains a valid *user identifier*.

✍ For operating systems that do not support the concept of a *user identifier* the implementation shall set this field to 2^{32} - 1 to indicate an invalid UID, unless otherwise specified by the user.

3.3.3.2 Uint32 Gid

☞ For operating systems that do not support the concept of a *group identifier* the implementation shall ignore this field. For operating systems that do support this field a value of 2^{32} - 1 shall indicate an invalid GID, otherwise the field contains a valid *group identifier*.

✍ For operating systems that do not support the concept of a *group identifier* the implementation shall set this field to 2^{32} - 1 to indicate an invalid GID, unless otherwise specified by the user.

3.3.3.3 Uint32 Permissions

```
/* Definitions: */
/*  Bit     for a File                  for a Directory             */
/*  ------- -----------------------     --------------------------- */
/*  Execute May execute file            May search directory        */
/*  Write   May change file contents    May create and delete files  */
/*  Read    May examine file contents   May list files in directory  */
/*  ChAttr  May change file attributes May change dir attributes    */
/*  Delete  May delete file             May delete directory        */

#define OTHER_Execute 0x00000001
#define OTHER_Write   0x00000002
#define OTHER_Read    0x00000004
#define OTHER_ChAttr  0x00000008
#define OTHER_Delete  0x00000010

#define GROUP_Execute 0x00000020
#define GROUP_Write   0x00000040
#define GROUP_Read    0x00000080
#define GROUP_ChAttr  0x00000100
#define GROUP_Delete  0x00000200

#define OWNER_Execute 0x00000400
#define OWNER_Write   0x00000800
#define OWNER_Read    0x00001000
#define OWNER_ChAttr  0x00002000
#define OWNER_Delete  0x00004000
```

The concept of permissions that deals with security is not completely portable between operating systems. This document attempts to maintain consistency among implementations in processing the permission bits by addressing the following basic issues:

1. How should an implementation handle Owner, Group and Other permissions when the operating system has no concept of User and Group Ids?
2. How should an implementation process permission bits when encountered, specifically permission bits that do not directly map to an operating system supported permission bit?
3. What default values should be used for permission bits that do not directly map to an operating system supported permission bit when creating a new file?

Owner, Group and Other

In general, for operating systems that do not support User and Group Ids the following algorithm should be used when processing permission bits:

When reading a specific permission, the logical OR of all three (owner, group, other) permissions should be the value checked. For example a file would be considered writable if the logical OR of OWNER_Write, GROUP_Write and OTHER_Write was equal to one.

When setting a specific permission the implementation should set all three (owner, group, other) sets of permission bits. For example to mark a file as writable the OWNER_Write, GROUP_Write and OTHER_Write should all be set to one.

Default Permission Values

For the operating systems covered by this document the following table describes what default values should be used for permission bits that do not directly map to an operating system supported permission bit when creating a new file.

Permission	File/Directory	Description	DOS	OS/2	Win 95	Win NT	Mac OS	UNIX & OS/400
Read	file	The file may be read	1	1	1	1	1	U
Read	directory	The directory may be read, only if the directory is also marked as *Execute*.	1	1	1	1	1	U
Write	file	The file's contents may be modified	U	U	U	U	U	U
Write	directory	Files or subdirectories may be renamed, added, or deleted, only if the directory is also marked as *Execute*.	U	U	U	U	U	U
Execute	file	The file may be executed.	0	0	0	0	0	U
Execute	directory	The directory may be searched for a specific file or subdirectory.	1	1	1	1	1	U
Attribute	file	The file's permissions may be changed.	1	1	1	1	1	Note 1
Attribute	directory	The directory's permissions may be changed.	1	1	1	1	1	Note 1
Delete	file	The file may be deleted.	Note 2	Note 2	Note 2	Note 2	Note 2	Note 2
Delete	directory	The directory may be deleted.	Note 2	Note 2	Note 2	Note 2	Note 2	Note 2

U - User Specified, 1 - Set, 0 - Clear

NOTE 1: Under UNIX only the owner of a file/directory may change its attributes. Under OS/400 if a file or directory is marked as writable (*Write* permission set) then the *Attribute* permission bit should be set.

NOTE 2: The *Delete* permission bit should be set based upon the status of the *Write* permission bit. Under DOS, OS/2 and Macintosh, if a file or directory is marked as writable (*Write* permission set) then the file is considered deletable and the *Delete* permission bit should be set. If a file is Read-Only then the *Delete* permission bit should not be set. This applies to file create as well as changing attributes of a file.

Processing Permissions

Implementation shall process the permission bits according to the following table that describes how to process the permission bits under the operating systems covered by this document. The table addresses the issues associated with permission bits that do not directly map to an operating system supported permission bit.

Permission	File/Directory	Description	DOS	OS/2	Win 95	Win NT	Mac OS	UNIX	OS/400
Read	file	The file may be read	E	E	E	E	E	E	E
Read	directory	The directory may be read	E	E	E	E	I	E	E
Write	file	The file's contents may be modified	E	E	E	E	E	E	E
Write	directory	Files or subdirectories may be created, deleted or renamed	E	E	E	E	E	E	E
Execute	file	The file may be executed.	I	I	I	I	I	E	I
Execute	directory	The directory may be searched for a specific file or subdirectory.	E	E	E	E	E	E	E
Attribute	file	The file's permissions may be changed.	E	E	E	E	E	I	I
Attribute	directory	The directory's permissions may be changed.	E	E	E	E	E	I	I
Delete	file	The file may be deleted.	E	E	E	E	E	I	I
Delete	directory	The directory may be deleted.	E	E	E	E	E	I	I

E - Enforce, I - Ignore

The *Execute* bit for a directory, sometimes referred to as the *search* bit, has special meaning. This bit enables a directory to be searched, but not have its contents listed. For example assume a directory called PRIVATE exists which only has the *Execute* permission and does not have the *Read* permission bit set. The contents of the directory PRIVATE can not be listed. Assume there is a file within the PRIVATE directory called README. The user can get access to the README file since the PRIVATE directory is searchable.

To be able to list the contents of a directory both the *Read* and *Execute* permission bits must be set for the directory. To be able to create, delete and rename a file or subdirectory both the *Write* and *Execute* permission bits must be set for the directory. To get a better understanding of the *Execute* bit for a directory reference any UNIX book that covers file and directory permissions. The rules defined by the *Execute* bit for a directory shall be enforced by all implementations. The exception to this rule applies to Macintosh implementations. A Macintosh implementation may ignore the status of the *Read* bit in determining the accessibility of a directory

NOTE 3: To be able to delete a file or subdirectory the *Delete* permission bit for the file or subdirectory must be set, and both the *Write* and *Execute* permission bits must be set for the directory it occupies.

3.3.3.4 Uint64 UniqueID

Section 3.2.1 describes how the value for this field is set. For file systems using a VAT, the function of the LVHD UniqueID field in the LVID is taken over by the VAT ICB File Entry UniqueID field, see 3.2.1.1.

NOTE: For UDF 2.00 and higher, the Unique ID value used in the UDF Unique ID Mapping Data is taken from the File Identifier Descriptor rather than from the File Entry.

3.3.3.5 byte ExtendedAttributes[]

Certain extended attributes should be recorded in this field of the *File Entry* for performance reasons. Other extended attributes should be recorded in an ICB pointed to by the field *Extended Attribute ICB*. In section 3.3.4 on *Extended Attributes* it will be specified which extended attributes should be recorded in this field.

3.3.4 Extended Attributes

In order to handle some of the longer Extended Attributes (EAs) that may vary in length, the following rules apply to the EA space.

1. *All* EAs with an attribute length greater than or equal to a logical block shall be block aligned by starting and ending on a logical block boundary. The one and only exception to this rule is the start of the first ECMA 167 EA.
2. Smaller EAs shall be constrained to an attribute length that is a multiple of 4 bytes.
3. Each Extended Attributes Space shall appear as a single contiguous logical space constructed as follows:

ECMA 167 EAs
Non block aligned Implementation Use EAs
Block aligned Implementation Use EAs
Application Use EAs

NOTE: There may exist 2 Extended Attributes Spaces per file, one embedded in the *File Entry* or *Extended File Entry* and the other as a separate space referenced by the Extended Attribute ICB address in the *File Entry* or *Extended File Entry*. Each Extended Attributes Space, if present, must have its own Extended Attribute Header Descriptor (see next section).

3.3.4.1 Extended Attribute Header Descriptor

```
struct ExtendedAttributeHeaderDescriptor {      /* ECMA 167 4/14.10.1 */
        struct tag      DescriptorTag;
        Uint32          ImplementationAttributesLocation;
        Uint32          ApplicationAttributesLocation;
}
```

☞ A value in one of the *location* fields highlighted above equal to or greater than the length of the EA space shall be interpreted as an indication that the corresponding attribute does not exist.

✍ If an attribute associated with one of the *location* fields highlighted above does not exist, then the value of the corresponding *location* field shall be set to #FFFFFFFF.

3.3.4.2 Alternate Permissions

```
struct AlternatePermissionsExtendedAttribute { /* ECMA 167 4/14.10.4 */
        Uint32          AttributeType;
        Uint8           AttributeSubtype;
        byte            Reserved[3];
        Uint32          AttributeLength;
        Uint16          OwnerIdentification;
        Uint16          GroupIdentification;
        Uint16          Permission;
}
```

This structure shall not be recorded.

3.3.4.3 File Times Extended Attribute

```
struct FileTimesExtendedAttribute {          /* ECMA 167 4/14.10.5 */
        Uint32          AttributeType;
        Uint8           AttributeSubtype;
        byte            Reserved[3];
        Uint32          AttributeLength;
        Uint32          DataLength;
        Uint32          FileTimeExistence;
        byte            FileTimes;
}
```

3.3.4.3.1 byte FileTimes

- If this field contains a file creation time it shall be interpreted as the creation time of the associated file. If the main *File Entry* is an *Extended File Entry*, the file creation time in this structure shall be ignored and the file creation time from the main *File Entry* shall be used.

- ✍ If the main File Entry is an Extended File Entry, this structure shall not be recorded with a file creation time.

If the main *File Entry* is not an *Extended File Entry* and the File Times Extended Attribute does not exist or does not contain the file creation time then an implementation shall use the *Modification Time* field of the *File Entry* to represent the file creation time.

3.3.4.4 Device Specification Extended Attribute

```
struct DeviceSpecificationExtendedAttribute {   /* ECMA 167 4/14.10.7 */
        Uint32          AttributeType;
        Uint8           AttributeSubtype;
        byte            Reserved[3];
        Uint32          AttributeLength;
        Uint32          ImplementationUseLength;   /* (=IU_L) */
        Uint32          MajorDeviceIdentification;
        Uint32          MinorDeviceIdentification;
        byte            ImplementationUse[IU_L];
}
```

The following paradigm shall be followed by an implementation that creates a *Device Specification Extended Attribute* associated with a file :

> If and only if a file has a *DeviceSpecificationExtendedAttribute* associated with it, the contents of the *File Type* field in the *icbtag* structure shall be set to 6 (indicating a block special device file), OR 7 (indicating a character special device file).
>
> If the contents of the *File Type* field in the *icbtag* structure do not equal 6 or 7, the *DeviceSpecificationExtendedAttribute* associated with a file shall be ignored.
>
> In the event that the contents of the *File Type* field in the *icbtag* structure equals 6 or 7, and the file does not have a *DeviceSpecificationExtendedAttribute* associated with it, access to the file shall be denied.
>
> For operating system environments that do not provide for the semantics associated with a block special device file, requests to open/read/write/close a file that has the *DeviceSpecificationExtendedAttribute* associated with it shall be denied.

3.3.4.4.1 ImplementationUse[IU_L]

As the first structure in the *ImplementationUse* field, an EntityID shall be recorded by all implementations. This EntityID uniquely identifies the current implementation by a Developer ID, see 2.1.5.

3.3.4.5 Implementation Use Extended Attribute

```
struct ImplementationUseExtendedAttribute {    /* ECMA 167 4/14.10.8 */
        Uint32              AttributeType;
        Uint8               AttributeSubtype;
        byte                Reserved[3];
        Uint32              AttributeLength;
        Uint32              ImplementationUseLength;   /* (=IU_L) */
        struct EntityID     ImplementationIdentifier;
        byte                ImplementationUse[IU_L];
}
```

The *AttributeLength* field specifies the length of the entire extended attribute. For variable length extended attributes defined using the *Implementation Use Extended Attribute* the *Attribute Length* field should be large enough to leave padding space between the end of the *Implementation Use* field and the end of the *Implementation Use Extended Attribute.*

The following sections describe how the *Implementation Use Extended Attribute* is used under various operating systems to store operating system specific extended attributes.

The structures defined in the following sections contain a *Header Checksum* field. This field represents a 16-bit checksum of the Implementation Use Extended Attribute header. The fields *AttributeType* through *ImplementationIdentifier* inclusively represent the data covered by the checksum. The *Header Checksum* field is used to aid in disaster recovery of the extended attributes space. C source code for the Header Checksum may be found in appendix 6.8.

NOTE: All compliant implementations shall preserve existing extended attributes encountered on the media. Implementations shall create and support the extended attributes for the operating system they currently support. For example, a Macintosh implementation shall preserve any OS/2 extended attributes encountered on the media. It shall also create and support all Macintosh extended attributes specified in this document.

3.3.4.5.1 All Operating Systems

3.3.4.5.1.1 FreeEASpace

This extended attribute shall be used to indicate unused space within the Extended Attributes Space. This extended attributes shall be stored as an *Implementation Use Extended Attribute* whose *ImplementationIdentifier* shall be set to:

"*UDF FreeEASpace"

The *ImplementationUse* area for this extended attribute shall be structured as follows:

FreeEASpace format

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	IU_L-2	Free EA Space	bytes

This extended attribute allows an implementation to shrink/grow the total size of other extended attributes without rewriting the complete Extended Attributes Space. The *FreeEASpace* extended attribute may be overwritten and the space re-used by any implementation that sees a need to overwrite it.

3.3.4.5.1.2 DVD Copyright Management Information

This extended attribute shall be used to store DVD Copyright Management Information. This extended attribute shall be stored as an *Implementation Use Extended Attribute* whose *ImplementationIdentifier* shall be set to:

"***UDF DVD CGMS Info**"

The *ImplementationUse* area for this extended attribute shall be structured as follows:

DVD CGMS Info format

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	1	CGMS Information	byte
3	1	Data Structure Type	Uint8
4	4	Protection System Information	bytes

This extended attribute allows DVD Copyright Management Information to be stored. The interpretation of this format shall be defined in the DVD specification published by the DVD Format/Logo Licensing Corporation, see 6.9.3. Support for this extended attribute is optional.

3.3.4.5.2 MS-DOS, Windows 95, Windows NT

👓 Ignored.

✍ Not supported. Extended attributes for existing files on the media shall be preserved.

3.3.4.5.3 OS/2

OS/2 supports an unlimited number of extended attributes, which shall be stored as a Named Stream as defined in 3.3.8.2. To enhance performance the following *Implementation Use Extended Attribute* will be created.

3.3.4.5.3.1 OS2EALength

This attribute specifies the OS/2 Extended Attribute Stream (3.3.8.2) information length. Since this value needs to be reported back to OS/2 under certain directory operations, for performance reasons it *should* be recorded in the *ExtendedAttributes* field of the *File Entry*. This extended attribute shall be stored as an *Implementation Use Extended Attribute* whose *ImplementationIdentifier* shall be set to:

"***UDF OS/2 EALength**"

The *ImplementationUse* area for this extended attribute shall be structured as follows:

OS2EALength format

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	4	OS/2 Extended Attribute Length	Uint32

The value recorded in the *OS2ExtendedAttributeLength* field shall be equal to the *Information Length* field of the File Entry for the ***OS2EA*** stream.

3.3.4.5.4 Macintosh OS

The Macintosh OS requires the use of the following extended attributes.

3.3.4.5.4.1 MacVolumeInfo

This extended attribute contains Macintosh volume information which shall be stored as an *Implementation Use Extended Attribute* whose *ImplementationIdentifier* shall be set to:

"***UDF Mac VolumeInfo**"

The *ImplementationUse* area for this extended attribute shall be structured as follows:

MacVolumeInfo format

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	12	Last Modification Date	Timestamp
14	12	Last Backup Date	Timestamp
26	32	Volume Finder Information	Uint32

The *MacVolumeInfo* extended attribute shall be recorded as an extended attribute of the root directory *File Entry*.

3.3.4.5.4.2 MacFinderInfo

This extended attribute contains Macintosh Finder information for the associated file or directory. Since this information is accessed frequently, for performance reasons it *should* be recorded in the *ExtendedAttributes* field of the *File Entry*.

The *MacFinderInfo* extended attribute shall be stored as an *Implementation Use Extended Attribute* whose *ImplementationIdentifier* shall be set to:

"***UDF Mac FinderInfo**"

The *ImplementationUse* area for this extended attribute shall be structured as follows:

MacFinderInfo format for a directory

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	2	Reserved for padding	Uint16 = 0
4	4	Parent Directory ID	Uint32
8	16	Directory Information	UDFDInfo
24	16	Directory Extended Information	UDFDXInfo

MacFinderInfo format for a file

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	2	Reserved for padding	Uint16 = 0
4	4	Parent Directory ID	Uint32
8	16	File Information	UDFFInfo
24	16	File Extended Information	UDFFXInfo
40	4	Resource Fork Data Length	Uint32
44	4	Resource Fork Allocated Length	Uint32

The *MacFinderInfo* extended attribute shall be recorded as an extended attribute of every file and directory within the Logical Volume.

The following structures used within the *MacFinderInfo* structure are listed below for clarity. For complete information on these structures refer to the Macintosh books called “Inside Macintosh”. The volume and page number listed with each structure correspond to a specific “Inside Macintosh” volume and page.

UDFPoint format (Volume I, page 139)

RBP	Length	Name	Contents
0	2	V	Int16
2	2	H	Int16

UDFRect format (Volume I, page 141)

RBP	Length	Name	Contents
0	2	Top	Int16
2	2	Left	Int16
4	2	Bottom	Int16
6	2	Right	Int16

UDFDInfo format (Volume IV, page 105)

RBP	Length	Name	Contents
0	8	FrRect	UDFRect
8	2	FrFlags	Int16
10	4	FrLocation	UDFPoint
14	2	FrView	Int16

UDFDXInfo format (Volume IV, page 106)

RBP	Length	Name	Contents
0	4	FrScroll	UDFPoint
4	4	FrOpenChain	Int32
8	1	FrScript	Uint8
9	1	FrXflags	Uint8
10	2	FrComment	Int16
12	4	FrPutAway	Int32

UDFFInfo format (Volume II, page 84)

RBP	Length	Name	Contents
0	4	FdType	Uint32
4	4	FdCreator	Uint32
8	2	FdFlags	Uint16
10	4	FdLocation	UDFPoint
14	2	FdFldr	Int16

UDFFXInfo format (Volume IV, page 105)

RBP	Length	Name	Contents
0	2	FdIconID	Int16
2	6	FdUnused	bytes
8	1	FdScript	Int8
9	1	FdXFlags	Int8
10	2	FdComment	Int16
12	4	FdPutAway	Int32

NOTE: The above-mentioned structures have their original Macintosh names preceded by “UDF” to indicate that they are actually different from the original Macintosh structures. On the media the UDF structures are stored *little endian* as opposed to the original Macintosh structures that are in *big endian* format.

3.3.4.5.5 UNIX

- 👓 Ignored.
- ✍ Not supported. Extended attributes for existing files on the media shall be preserved.

3.3.4.5.6 OS/400

OS/400 requires the use of the following extended attributes.

3.3.4.5.6.1 OS400DirInfo

This attribute specifies the OS/400 extended directory information. Since this value needs to be reported back to OS/400 for normal directory information processing, for performance reasons it should be recorded in the ExtendedAttributes field of the File Entry. This extended attribute shall be stored as an Implementation Use Extended Attribute whose ImplementationIdentifier shall be set to:

"***UDF OS/400 DirInfo**".

The *ImplementationUse* area for this extended attribute shall be structured as follows:

OS400DirInfo format

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	2	Reserved for padding	Uint16 = 0
4	44	DirectoryInfo	bytes

For complete information on the structure of the *DirectoryInfo* field recorded in the *OS400DirInfo* format, refer to the following IBM document:

IBM OS/400 UDF Implementation
Optical Storage Solutions, Department HTT
IBM
Rochester, Minnesota

3.3.4.6 Application Use Extended Attribute

```
struct ApplicationUseExtendedAttribute {        /* ECMA 167 4/14.10.9 */
        Uint32          AttributeType;      /* = 65536 */
        Uint8           AttributeSubtype;
        byte            Reserved[3];
        Uint32          AttributeLength;
        Uint32          ApplicationUseLength;   /* (=AU_L) */
        struct EntityID ApplicationIdentifier;
        byte            ApplicationUse[AU_L];
}
```

The *AttributeLength* field specifies the length of the entire extended attribute. For variable length extended attributes defined using the *Application Use Extended Attribute* the *Attribute Length* field should be large enough to leave padding space between the end of the *ApplicationUse* field and the end of the *Application Use Extended Attribute.*

The structures defined in the following section contain a *Header Checksum* field. This field represents a 16-bit checksum of the Application Use Extended Attribute header. The fields *AttributeType* through *ApplicationIdentifier* inclusively represent the data covered by the checksum. The Header *Checksum* field is used to aid in disaster recovery of the extended attributes space. C source code for the Header Checksum may be found in appendix 6.8.

NOTE: All compliant implementations shall preserve existing extended attributes encountered on the media. Implementations shall create and support the extended attributes for the operating system they currently support. For example, a Macintosh implementation shall preserve any OS/2 extended attributes encountered on the media. It shall also create and support all Macintosh extended attributes specified in this document.

3.3.4.6.1 All Operating Systems

3.3.4.6.1.1 FreeAppEASpace

This extended attribute shall be used to indicate unused space within the Extended Attributes Space reserved for Application Use Extended Attributes. This extended attribute shall be stored as an *Application Use Extended Attribute* whose *ApplicationIdentifier* shall be set to:

"***UDF FreeAppEASpace**"

The *ApplicationUse* area for this extended attribute shall be structured as follows:

FreeAppEASpace format

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	IU_L-2	Free EA Space	bytes

This extended attribute allows an implementation to shrink/grow the total size of other extended attributes without rewriting the complete Extended Attributes Space. The *FreeAppEASpace* extended attribute may be overwritten and the space re-used by any implementation who sees a need to overwrite it.

3.3.5 Named Streams

Named Streams provide a mechanism for associating related data of a file. It is similar in concept to extended attributes. However, Named Streams have significant advantages over extended attributes. They are not as limited in length. Space management is much easier as each Named Stream has its own space, rather than the common space of extended attributes. Finding a particular Named Stream does not involve searching the entire data space, as it does for extended attributes.

Named Streams are mainly intended for user data. For example, a database application may store the records in the default or main stream and indices in Named Streams. The user would then see only one file for the database rather than many, and the application can use the various Named Streams almost as if they were independent files.

Named Streams are identified by an Extended File Entry. Extended File Entries are required for files with associated Named Streams. Files without Named Streams should use Extended File Entries. Files may have normal File Entries; normal File Entries would be used where backward compatibility is desired, such as writing DVD Video discs.

There is a "*System Stream Directory*" which is the stream directory identified by the File Set Descriptor. These streams are used to describe data related to the entire medium instead of data that relates to a file. UDF defines several "*System Streams*" that are to be identified by the *System Stream Directory*.

The parent of the *System Stream Directory* shall be the *System Stream Directory*.

It is recommended that Named Streams be used to store metadata and application data instead of Extended Attributes in new implementations.

3.3.5.1 Named Streams Restrictions

ECMA 167 3rd edition defines a new Extended File Entry that contains a field for identifying a Stream Directory. This new Extended File Entry should be used in place of the old File Entry, and should be used for describing the Named Streams themselves. File Entries and Extended File Entries may be freely mixed. In particular, compatibility with old reader implementations can be maintained for certain files.

Restrictions:

The Stream Directory ICB field of ICBs describing Stream Directories or Named Streams shall be set to zero. [no hierarchical streams]

Each Named Stream shall be identified by exactly one FID in exactly one Stream Directory. [no hard links among Named Streams or files and Named Streams]

Each Stream Directory ICB shall be identified by exactly one Stream Directory ICB field. [no hard links to Stream Directories]. The sole exception is that the parent of the System Stream Directory shall be the System Stream Directory.

Hard Links to files with Named Streams are allowed.

Named Streams and Stream Directories shall not have Extended Attributes.

Section 3.2.1.1 describes how the Unique ID fields of File Identifier Descriptors and File Entries/Extended File Entries defining Named Streams and Stream Directories are set.

The UID, GID, and permissions fields of the main File Entry shall apply to all Named Streams associated with the main stream. At the time of creation of a Named Stream the values of the UID, GID and permissions fields of the main File Entry should be used as the default values for the corresponding fields of the Named Stream. Implementations are not required to maintain or check these fields in a Named Stream.

Implementations should not present Named Streams marked with the *metadata* bit set in the FID to the user. Named Streams marked with the *metadata* bit are intended solely for the use of the file system implementation.

The parent entry FID in a Stream Directory points to the main Extended File Entry, so its reference must be counted in the File Link Count field of the Extended File Entry. The sole exception is that the parent of the System Stream Directory shall be the System Stream Directory.

NOTE: There is a potential pitfall when deleting files/directories: if the File Link Count goes to one when a FID is deleted, implementations must check for the presence of a Stream Directory. If present, there are no more FIDs pointing to this File Entry, so it and all associated structures must be deleted.

The modification time field of the main Extended File Entry should be updated whenever any associated named stream is modified. The Access Time field of the main Extended File Entry should be updated whenever any associated named stream is accessed. The SETUID and SETGID bits of the ICB Tag flags field in the main Extended File Entry should be cleared whenever any associated named stream is modified.

The ICB for a Named Stream directory shall have a file type of 13. All Named Streams shall have a file type of 5.

All systems shall make the main data stream available, even on implementations that do not implement Named Streams.

3.3.5.2 UDF Defined Named Streams (Metadata)

A set of Named Streams is defined by UDF for file system use. Some UDF Named Streams are identified by the File Set Descriptor (*System Stream Directory*) and apply to the entire file set. These are called UDF Defined System Streams and are defined in section 3.3.7. Others pertain to individual files or directories and are identified by the Stream Directory of that particular file or directory. These are called UDF Defined Non-System Streams and are defined in 3.3.8.

All UDF Defined Named Streams shall have the Metadata bit set in the File Identifier Descriptor in the Stream Directory, unless otherwise specified in this document. All Named Streams not generated by the file system implementation shall have this bit set to zero.

The four characters *UDF are the first four characters of all UDF defined named streams in this document. Implementations shall not use any identifier beginning with *UDF for named streams that are not defined in this document. All identifiers for named streams beginning with *UDF are reserved for future definition by OSTA.

3.3.6 Extended Attributes as Named Streams

NOTE: Because conversion of some types of Extended Attributes to a Named Stream appeared to be impossible and because it was never intended to allow automatic conversion of any EA to a Named Stream, this section is amended for UDF revisions after UDF 2.01. Conversion of any EA to a Named Stream is not allowed.

3.3.7 UDF Defined System Streams

This section contains the definition of UDF defined System Streams.

Stream Name	Stream Location	Metadata Flag
"*UDF Unique ID Mapping Data"	System Stream Directory (File Set Descriptor)	1
"*UDF Non-Allocatable Space"	System Stream Directory (File Set Descriptor)	1
"*UDF Power Cal Table"	System Stream Directory (File Set Descriptor)	1
"*UDF Backup"	System Stream Directory (File Set Descriptor)	1

Since the System Streams listed above have the Metadata flag set, the implementation shall not pass the name of the System Stream across the "plug-in file system interface" of a platform.

3.3.7.1 Unique ID Mapping Data Stream

The Unique ID Mapping Data allows an implementation to go directly to the ICB hierarchy for the file/directory associated with a UDF UniqueID, or to the ICB hierarchy for the directory that contains the file/directory associated with the UDF UniqueID. Note that for UDF release 2.00 and higher the UDF UniqueID value used for this purpose is taken from the File Identifier Descriptor rather than from the File Entry.

Unique ID Mapping Data is stored as a Named Stream of the *System Stream Directory* (associated with the File Set Descriptor). The name of this System Stream shall be set to:

"***UDF Unique ID Mapping Data**"

The *Metadata* bit in the *File Characteristics* field of the File Identifier Descriptor for the stream shall be set to 1 to indicate that the existence of this stream should not be made known to clients of a platform's file system interface.

Rules for the presence and consistency of the Unique ID Mapping Data Stream:

- Shall be created for Read-Only media
- Shall be created by implementations with batch write (e.g., pre-mastering tools) a volume on Write-Once and Rewritable media

For implementations which perform incremental updates of volumes on Write-Once or Rewritable media (e.g., on-line file systems), the following rules apply:

- May be created and maintained if not present
- Shall be maintained if present and volume is clean
- Should be repaired and maintained, but may be deleted, if present and volume is dirty

For these rules, a volume is clean if either a valid Close Logical Volume Integrity Descriptor or a valid Virtual Allocation Table is recorded.

3.3.7.1.1 UDF Unique ID Mapping Data

The contents of the Unique ID Mapping Stream are described by the tables "UDF Unique ID Mapping Data" and "UDF Unique ID Mapping Entry". The mapping data contains some header fields before an array of mapping entries. The fields of these structures are described below their corresponding table.

UDF Unique ID Mapping Data

RBP	Length	Name	Contents
0	32	Implementation Identifier	EntityID
32	4	Flags	Uint32
36	4	Mapping Entry Count (=MEC)	Uint32
40	8	Reserved	Bytes (= #00)
48	16*MEC	Mapping Entries	IDMappingEntry

Implementation Identifier is described in section 2.1.5.

Flags are defined as follows:

Bit 0	Index Bit
Bits 1 – 31	Reserved, shall be set to ZERO

Index Bit set to ONE is called Index Mode. In Index Mode, the UDF UniqueID, once decremented by 16 (the value Next UniqueID is initialized to), can be used as an index into the array Mapping Entries.

Mapping Entry Count is the size, in entries, of the array Mapping Entries.

Mapping Entries is an array of UDF Unique ID Mapping Entry structures. There is one mapping entry for every non-stream, non-parent File Identifier Descriptor. Whenever the volume is consistent, the array is always sorted in ascending order of UDF UniqueID.

3.3.7.1.2 UDF Unique ID Mapping Entry

UDF Unique ID Mapping Entry

RBP	Length	Name	Contents
0	4	UDF Unique ID	Uint32
4	4	Parent Logical Block Number	Uint32
8	4	Object Logical Block Number	Uint32
12	2	Parent Partition Reference Number	Uint16
14	2	Object Partition Reference Number	Uint16

UDF Unique ID is the value found in the FID identifying the object.

Parent Logical Block Number is the logical block number of the ICB identifying the directory that contains the FID identifying the object.

Object Logical Block Number is the logical block number from the long_ad ICB field of the FID identifying the object.

Parent Partition Reference Number is the partition reference number of the ICB identifying the directory that contains the FID identifying the object.

Object Partition Reference Number is the partition reference number from the long_ad ICB field of the FID identifying the object.

In Index Mode, the first entry has a UDF Unique ID of 16 and subsequent entries are required to have a UDF Unique ID value of one more than the preceding entry.

If not in Index Mode, invalid entries may be removed in order to shrink the array. Invalid entries are represented by having a value of zero in all fields, except the UDF Unique ID field. Invalid entries are the result of objects that were deleted from the medium or entries at the end of the Mapping Entries array that are not yet in use.

There shall only be valid entries for non-stream, non-parent FIDs.

NOTE: The UDF Unique ID value of a mapping entry for an object needs not be equal to the Unique ID value found in the File Entry of the object.

The correctness of a mapping entry can be verified performing the following steps:

1. Read the File Entry of the parent directory of the object using the Parent Logical Block Number and the Parent Partition Reference Number of the mapping entry.
2. Find in the parent directory a FID with a UDF Unique ID value equal to the UDF Unique ID of the mapping entry.
3. The long_ad ICB field of this FID shall contain logical block number and partition reference number values equal to the Object Logical Block Number and Object Partition Reference Number values of the mapping entry respectively.

3.3.7.2 Non-Allocatable Space Stream

ECMA 167 does not provide for a mechanism to describe defective areas on media or areas not usable due to allocation outside of the file system. The *Non-Allocatable Space Stream* provides a method to describe space not usable by the file system. The *Non-Allocatable Space Stream* shall be recorded only on volumes with a Sparable Partition Map recorded.

The *Non-Allocatable Space Stream* shall be generated at format time. All space indicated by the *Non-Allocatable Space Stream* shall also be marked as allocated in the Unallocated Space Bitmap or Table. The *Non-Allocatable Space Stream* shall be recorded as a System Stream in the System Stream Directory of the *File Set Descriptor*. The System Stream name shall be:

"***UDF Non-Allocatable Space**"

The stream shall be marked with the attributes *Metadata* (bit 4 of file characteristics set to ONE) and System (bit 10 of ICB Tag flags field set to ONE). The stream's Allocation Descriptors shall identify all non-allocatable packets. The Allocation Descriptors shall have allocation type 1 (allocated but not recorded). The Information Length in the File Entry of this stream shall be zero; so all Allocation Descriptors are in the file tail. This stream shall include both defective packets found at format time and space allocated for sparing at format time.

3.3.7.3 Power Calibration Stream

One of the potential limitations on the effective use of the packet-write capabilities of CD-Recordable drives is the limited number (100) of power calibration areas available on current CD-R media. These power calibration areas are used to establish the appropriate power calibration settings with which data can be successfully and reliably written to the CD-R disc currently in the drive. The appropriate settings for a specific drive can vary significantly from disc to disc, between two different drives of the same make and model, and even using the same disc, drive and system configuration, but under different environmental conditions.

Because of this, most current CD-R drives recalibrate themselves the first time a write is attempted after a media change has occurred. This imposes no restriction on recording to discs using the disc-at-once or track-at-once modes, since in each of these modes the disc will fill (either by consuming the total available data capacity or total number of recordable tracks) in less than 100 separate writes. When using packet-write though, the disc could be written to thousands of times over an extended period before the disc is full.

Suppose, for instance, one wanted to incrementally back-up any new and/or modified files at the end of each work day (though the drive might also be used intermittently to do other projects during the day). These back-ups may require writing as little as a megabyte (or even less) each day. If one of the power calibration areas is used to calibrate the drive before writing to the disc every day, within five months the power calibration areas will all have been used, but only a small fraction of the total disc capacity will have been consumed. It is likely that such a result would be both unexpected and unacceptable to the user of such a product.

The industry is attempting to provide ways to reduce the frequency with which the power calibration area of a CD-Recordable disc must be used. At least one current CD-R drive model tries to remember the power calibration values last used for recording data on each of a small number of recently encountered discs. Most CD-Recordable drives provide a mechanism for the host software to retrieve from the drive the most recent power calibration settings used by the drive to record data on the current disc, and to restore and use such information at some future time.

The Power Calibration Table described herein would be used to store on the disc the power calibration information thus obtained for future use by compatible implementations. The table consists of a header followed by a list of records containing power calibration settings which have been used by various drives and/or hosts, under various conditions, to record data on this disc, as well as other relevant information which may be used to determine which of the recorded calibration settings may be appropriate for use in a future situation. While every effort has been made to anticipate and include all necessary information to make effective use of the recorded power calibration information possible, it is up to the individual implementation to determine if, when and how such information will actually be used.

The Power Calibration Table may be recorded as a System Stream of the File Set Descriptor according to the rules of 3.3.5. The name of the System Stream shall be as follows:

"***UDF Power Cal Table**"

Implementations that do not support the Power Calibration Table shall not delete this System Stream. Further, any implementation which supports and/or uses the Power Calibration Table shall not delete or modify any records from such table which the implementation, through its use thereof, did not clearly and specifically obsolete or update.

The stream shall be formatted as follows:

3.3.7.3.1 Power Calibration Table Stream

RBP	Length	Name	Contents
0	32	Implementation Identifier	EntityID [UDF 2.1.5]
32	4	Number of Records	Uint32 [1/7.1.5]
36	*	Power Calibration Table Records	bytes

Implementation Identifier:
See UDF section 2.1.5.

Number of Records:
Shall specify the number of records contained in the power calibration table

Power Calibration Table Records:
A series of power calibration table records for drives which have written to this disc. The length of this table is variable, but shall be a multiple of four bytes. Recording of data in any unstructured field shall be left justified and padded on the right with #20 bytes.

Power Calibration Table Record Layout

RBP	Length	Name	Contents
0	2	Record Length	Uint16 [1/7.1.3]
2	2	Drive Unique Area Length [DUA_L]	Uint16 [1/7.1.3]
4	32	Vendor ID	bytes
36	16	Product ID	bytes
52	4	Firmware Revision Level	bytes
56	16	Serial Number/Device Unique ID	bytes
72	8	Host ID	bytes
80	12	Originating TimeStamp	Timestamp [1/7.3]
92	12	Updated TimeStamp	Timestamp [1/7.3]
104	2	Speed	Uint16 [1/7.1.3]
106	6	Power Calibration Values	bytes
112	[DUA_L]	Drive Unique Area	bytes

Record Length – The length of this Power Calibration Table Record in bytes, including the optional variable length Drive Unique Area. Shall be a multiple of four bytes.

Drive Unique Area Length – The length of the optional Drive Unique Area recorded at the end of this record in bytes. Shall be a multiple of four bytes.

Vendor ID – The Vendor ID reported by the drive.

Product ID – The Product ID reported by the drive.

Firmware Revision Level – The Firmware Revision Level reported by the drive.

Serial Number/Device Unique ID – A serial number or other unique identifier for the specific drive, of the model specified by the vendor and product Ids given, which has successfully used the power calibration values reported herein to record data on this disc.

Host ID – The host serial number, ethernet ID, or other value (or combination of values) used by an implementation to identify the specific host computer to which the drive was attached when it successfully used the power calibration values reported herein to record data on this disc. An implementation shall attempt to provide a unique value for each host, but is not required to guarantee the value's uniqueness.

Originating TimeStamp – The date and time at which the power calibration values recorded herein were initially verified to have been successfully used.

Updated TimeStamp – The date and time at which the power calibration values recorded herein were most recently verified to have been successfully used.

Speed – The recording speed, as reported by the drive, at which power calibration values recorded herein were successfully used. This value is the number of kilobytes per second (bytes per second / 1000) that the data was written to the disc by the drive (truncating any fractions). For example, a speed of 176 means data was written to the disc at 176 Kbytes/second, which is the basic CD-DA (Digital Audio) data rate (a.k.a. "1X" for CD-DA). A speed of 353 means data was written to the disc at 353 Kbytes/second, or twice the basic CD-DA data rate (a.k.a. "2X" for CD-DA). CD-ROM recording rates should be adjusted upward (roughly 15%) to the corresponding CD-DA rates to determine the correct speed value (e.g. A "1X" CD-ROM data rate should be recorded as a "1X" CD-DA, which is a speed of 176). Note that these are raw data rates and do not reflect all overhead resulting from (additional) headers, error correction data, etc.

Power Calibration Values – The vendor-specific power calibration values reported by the drive.

Drive Unique Area – Optional area for recording unrestricted information unique to the drive (such as drive operating temperature), which certain implementations may use to enhance the use of the recorded power calibration information or the operation of the associated drive. The drive manufacturer shall define recording of data in this field. This area shall be an integral multiple of four bytes in length.

3.3.7.4 UDF Backup Time

The name of this System Stream shall be set to:

"***UDF Backup**"

This stream shall have the following contents, which should be embedded in the ICB:

UDF Backup Time

RBP	Length	Name	Contents
0	12	Backup Time	Timestamp

Backup Time is the latest time that a backup of this volume was performed.

3.3.8 UDF Defined Non-System Streams

This section defines the following non-system streams:

Stream Name	Stream Location	Metadata Flag
"*UDF Macintosh Resource Fork"	Any file	0
"*UDF OS/2 EA"	Any file or directory	0
"*UDF NT ACL"	Any file or directory	0
"*UDF UNIX ACL"	Any file or directory	0

3.3.8.1 Macintosh Resource Fork Stream

Because the Resource Fork is referenced by an explicit interface, UDF implementations are not provided the authoritative name for this stream. For the purpose of interchange, the name shall be set to:

"***UDF Macintosh Resource Fork**"

The *Metadata* bit in the *File Characteristics* field of the File Identifier Descriptor shall be set to 0 to indicate that the existence of this file should be made known to clients of a platform's file system interface.

3.3.8.2 OS/2 EA Stream

All OS/2 definable extended attributes shall be stored as a Named Stream whose name shall be set to:

"***UDF OS/2 EA**"

The *OS2EA Stream* contains a table of OS/2 Full EAs (*FEA)* as shown below.

FEA format

RBP	Length	Name	Contents
0	1	Flags	Uint8
1	1	Length of Name (=L_N)	Uint8
2	2	Length of Value (=L_V)	Uint16
4	L_N	Name	bytes
4+L_N	L_V	Value	bytes

For a complete description of Full EAs (*FEA*) please reference the following IBM document:

"Installable File System for OS/2 Version 2.0"
OS/2 File Systems Department
PSPC Boca Raton, Florida
February 17, 1992

3.3.8.3 Access Control Lists

Certain operating systems support the concept of Access Control Lists (ACLs) for enforcing file access restrictions. In order to facilitate support for ACL's UDF has defined a set of system level Named Streams, whose purpose is to store the ACL associated with a given file object.

ACLs under UDF are stored as Named Streams, following the rules of section 3.3.5. The contents of the Named Stream ACL shall be opaque and are not defined by this document. Interpretation of the contents of the named ACL shall be left to the operating system for which the ACL is intended. The following names shall be used to identify the ACLs and shall be reserved. These names shall not be used for application named streams.

"***UDF NT ACL**"

This name shall identify the named stream ACL for the Windows NT operating system.

"***UDF UNIX ACL**"

This name shall identify the named stream ACL for the UNIX operating system.

4. User Interface Requirements

4.1 Part 3 - Volume Structure

Part 3 of ECMA 167 contains various Identifiers that - depending upon the implementation - may have to be presented to the user.

- *VolumeIdentifier*
- *VolumeSetIdentifier*
- *LogicalVolumeIdentifier*
- *FileSetIdentifier*

These identifiers, which are stored in CS0, may have to go through some form of translation to be displayable to the user. Therefore when an implementation must perform an OS specific translation on the above listed identifiers the implementation shall use the algorithms described in section 4.2.2.1.

C source code for the translation algorithms is found in appendix 6.7 of this document.

4.2 Part 4 - File Structure

4.2.1 ICB Tag

```
struct icbtag {             /* ECMA 167 4/14.6 */
        Uint32      PriorRecordedNumberofDirectEntries;
        Uint16      StrategyType;
        byte        StrategyParameter[2];
        Uint16      MaximumNumberofEntries;
        byte        Reserved;  /* == #00 */
        Uint8       FileType;
        Lb_addr     ParentICBLocation;
        Uint16      Flags;
}
```

4.2.1.1 FileType

Any open/close/read/write requests for file(s) that have any of the following values in this field shall result in an *Access Denied* error condition under non-UNIX operating system environments:

> *File Type* values – 0 (Unknown), 6 (block device), 7 (character device), 9 (FIFO), and 10 (C_ISSOCK).

Any open/close/read/write requests to a file of type 12 (*Symbolic Link*) shall access the file/directory to which the symbolic link is pointing.

4.2.2 File Identifier Descriptor

```
struct FileIdentifierDescriptor {                /* ECMA 167 4/14.4 */
        struct tag          DescriptorTag;
        Uint16              FileVersionNumber;
        Uint8               FileCharacteristics;
        Uint8               LengthofFileIdentifier;
        struct long_ad      ICB;
        Uint16              LengthofImplementationUse;
        byte                ImplementationUse[];
        char                FileIdentifier[];
        byte                Padding[];
}
```

4.2.2.1 char FileIdentifier[]

Since most operating systems have their own specifications as to characteristics of a legal *FileIdentifier*, this becomes a problem with interchange. Therefore since all implementations must perform some form of *FileIdentifier* translation it would be to the users advantage if all implementations used the same algorithm.

The problems with *FileIdentifier* translations fall within one or more of the following categories:

- *Name Length* - Most operating systems have some fixed limit for the length of a File Identifier.
- *Invalid Characters* - Most operating systems have certain characters considered as being illegal within a file identifier name.
- *Displayable Characters* - Since UDF supports the Unicode character set standard characters within a file identifier may be encountered which are not displayable on the receiving system.
- *Case Insensitive* - Some operating systems are case insensitive in regards to file identifiers. For example OS/2 preserves the original case of the file identifier when the file is created, but uses a case insensitive operations when accessing the file identifier. In OS/2 "Abc" and "ABC" would be the same file name.
- *Reserved Names* - Some operating systems have certain names that cannot be used for a file identifier name.

The following sections outline the *FileIdentifier* translation algorithm for each specific operating system covered by this document. This algorithm shall be used by all OSTA UDF compliant implementations. The algorithm *only applies when reading* an illegal *FileIdentifier*. The original *FileIdentifier* name on the media should not be modified. This algorithm shall be applied by any implementation that performs some form of *FileIdentifier* translation to meet operating system file identifier restrictions.

All OSTA UDF compliant implementations shall support the UDF translation algorithms, but may support additional algorithms. If multiple algorithms are supported the user of the implementation shall be provided with a method to select the UDF translation algorithms. It is recommended that the default displayable algorithm be the UDF defined algorithm.

The primary goal of these algorithms is to produce a *unique* file name that meets the specific operating system restrictions without having to scan the entire directory in which the file resides.

C source code for the following algorithms may be found in appendix 6.7 of this document.

NOTE 1: In the definition of the following algorithms anytime a d-character is specified in quotes, the Unicode hexadecimal value will also be specified. The following algorithms reference “CS0 Hex representation”, which corresponds to using the Unicode values #0030 - #0039, and #0041 - #0046 to represent a value in hex. In addition, the following algorithms reference “CS0 Base41 representation”, which corresponds to augmenting the CS0 Hex representation to use #0047 - #005A, #0023, #005F, #007E, #002D and #0040 to represent digits 16-40.

The following algorithms could still result in name-collisions being reported to the user of an implementation. However, the rationale includes the need for efficient access to the contents of a directory and consistent name translations across logical volume mounts and file system driver implementations, while allowing the user to obtain access to any file within the directory (through possibly renaming a file).

Some name transformations in section 4.2.2.1 result in two namespaces being visible at once in a given directory – the space of primary names, those which are physically recorded in a directory; and the space of generated names, those which are derived from the primary names. This is distinct from transformations that take an otherwise illegal name and render it into a legal form, the illegal name not being considered part of the namespace of the directory on that system. For UDF implementations using such transforms, the implementation should search a directory in two passes: pass one should match against the primary namespace and pass two should match against the generated namespace. A match in the primary namespace should be preferred to a match against the generated namespace.

Definitions:
A *FileIdentifier* shall be considered as being composed of two parts, a *file name* and *file extension.*

The character ‘.’ (#002E) shall be considered as the separator for the *FileIdentifier* of a file; characters appearing subsequent to the last ‘.’ (#002E) shall be considered as constituting the *file extension* if and only if it is less than or equal to 5 characters in length, otherwise the *file extension* shall not exist. Characters appearing prior to the *file extension*, excluding the last '.' (#002E), shall be considered as constituting the *file name*.

NOTE 2: Even though OS/2, Macintosh, and UNIX do not have an official concept of a filename extension it is common file naming conventions to end a file with "." Followed by a 1 to 5 character extension. Therefore the following algorithms attempt to preserve the *file extension* up to a maximum of 5 characters.

4.2.2.1.1 MS-DOS

Due to the restrictions imposed by the MS DOS operating system environments on the *FileIdentifier* associated with a file the following methodology shall be employed to handle *FileIdentifier(s)* under the above-mentioned operating system environments.

Exception: Implementations on non-MS-DOS systems that may normally provide dual namespaces (8.3 and non-8.3) using this transformation may omit or provide a mechanism for disabling its use.

Restrictions: The *file name* component of the *FileIdentifier* shall not exceed 8 characters. The *file extension* component of the *FileIdentifier* shall not exceed 3 characters.

1. *FileIdentifier* Lookup: Upon request for a *"lookup"* of a *FileIdentifier*, a case-insensitive comparison shall be performed.
2. Validate *FileIdentifier*: If the *FileIdentifier* is a valid MS-DOS file identifier then do not apply the following steps.
3. Remove Spaces: All embedded spaces within the identifier shall be removed.
4. Invalid Characters: A *FileIdentifier* that contains characters considered invalid within a *file name* or *file extension* (as defined above), or not displayable in the current environment, shall have them translated into "_" (#005F). (the File Identifier on the media is NOT modified). Multiple sequential invalid or non-displayable characters shall be translated into a single "_" (#005F) character. Reference appendix 6.7.1 on invalid characters for a complete list.
5. Leading Periods: In the event that there do not exist any characters prior to the first *"."* (#002E) character, leading *"."* (#002E) characters shall be disregarded up to the first non "." (#002E) character, in the application of this heuristic.
6. Multiple Periods: In the event that the *FileIdentifier* contains multiple *"."* (#002E) characters, all characters appearing subsequent to the last '.' (#002E) shall be considered as constituting the *file extension* if and only if it is less than or equal to 5 characters in length, otherwise the *file extension* shall not

exist. Characters appearing prior to the *file extension*, excluding the last '.' (#002E), shall be considered as constituting the *file name*. All embedded *"."* (#002E) characters within the *file name* shall be removed.

7. Long Extension: In the event that the number of characters constituting the *file extension* at this step in the process is greater than 3, the *file extension* shall be regarded as having been composed of the first 3 characters amongst the characters constituting the *file extension* at this step in the process.

8. Long Filename: In the event that the number of characters constituting the file name at this step in the process is greater than 8, the *file name* shall be truncated to 4 characters.

9. FileIdentifier CRC: Since through the above process character information from the original *FileIdentifier* is lost the chance of creating a duplicate *FileIdentifier* in the same directory increases. To greatly reduce the chance of having a duplicate *FileIdentifier* the file name shall be modified to contain a CRC of the original *FileIdentifier*. The *file name* shall be composed of the first 4 characters constituting the *file name* at this step in the process, followed by the separator '#' (#0023), followed by the 3 digit CS0 Base41 representation of the 16-bit CRC of the UNICODE expansion of the original filename.

10. The new file identifier shall be translated to all upper case.

4.2.2.1.2 OS/2

Due to the restrictions imposed by the OS/2 operating system environment, on the *FileIdentifier* associated with a file the following methodology shall be employed to handle *FileIdentifier(s)* under the above-mentioned operating system environment:

1. *FileIdentifier* Lookup: Upon request for a *"lookup"* of a *FileIdentifier*, a case-sensitive comparison may be performed. If the case-sensitive comparison is not done or if it fails, a case-insensitive comparison shall be performed.

2. Validate *FileIdentifier*: If the *FileIdentifier* is a valid OS/2 file identifier then do not apply the following steps.

3. Invalid Characters: A *FileIdentifier* that contains characters considered invalid within an OS/2 file name, or not displayable in the current environment shall have them translated into "_" (#005F). Multiple sequential invalid or non-displayable characters shall be translated into a single "_" (#005F) character. Reference appendix 6.7.2 on invalid characters for a complete list.

4. Trailing Periods and Spaces: All trailing "." (#002E) and " " (#0020) shall be removed.

5. FileIdentifier CRC: Since through the above process character information from the original *FileIdentifier* is lost the chance of creating a duplicate

FileIdentifier in the same directory increases. To greatly reduce the chance of having a duplicate *FileIdentifier* the *file name* shall be modified to contain a CRC of the original *FileIdentifier*.

If there is a *file extension* then the new *FileIdentifier* shall be composed of up to the first (254 – (length of (new *file extension*) + 1 (for the '.')) – 5 (for the #CRC)) characters constituting the *file name* at this step in the process, followed by the separator '#' (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*, followed by '.' (#002E) and the *file extension* at this step in the process.

Otherwise if there is no *file extension* the new *FileIdentifier* shall be composed of up to the first (254 – 5 (for the #CRC)) characters constituting the *file name* at this step in the process. Followed by the separator '#' (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*.

4.2.2.1.3 Macintosh

Due to the restrictions imposed by the Macintosh operating system environment, on the *FileIdentifier* associated with a file the following methodology shall be employed to handle *FileIdentifier(s)* under the above-mentioned operating system environment:

1. *FileIdentifier* Lookup: Upon request for a *"lookup"* of a *FileIdentifier*, a case-sensitive comparison may be performed. If the case-sensitive comparison is not done or if it fails, a case-insensitive comparison shall be performed.
2. Validate *FileIdentifier*: If the *FileIdentifier* is a valid Macintosh file identifier then do not apply the following steps.
3. Invalid Characters: A *FileIdentifier* that contains characters considered invalid within a Macintosh file name, or not displayable in the current environment, shall have them translated into "_" (#005F). Multiple sequential invalid or non-displayable characters shall be translated into a single "_" (#005F) character. Reference appendix 6.7.2 on invalid characters for a complete list
4. Long FileIdentifier: In the event that the number of characters constituting the *FileIdentifier* at this step in the process is greater than 31 (maximum name length for the Macintosh operating system), the new *FileIdentifier* will consist of the first *26* characters of the *FileIdentifier* at this step in the process.
5. FileIdentifier CRC: Since through the above process character information from the original *FileIdentifier* is lost the chance of creating a duplicate *FileIdentifier* in the same directory increases. To greatly reduce the chance of having a duplicate *FileIdentifier* the *file name* shall be modified to contain a CRC of the original *FileIdentifier*.

If there is a *file extension* then the new *FileIdentifier* shall be composed of up to the first (31 – (length of (new *file extension*) + 1 (for the '.')) – 5 (for the #CRC)) characters constituting the *file name* at this step in the process, followed by the separator '#' (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*, followed by '.' (#002E) and the *file extension* at this step in the process.

Otherwise if there is no *file extension* the new *FileIdentifier* shall be composed of up to the first (31 – 5(for the #CRC)) characters constituting the *file name* at this step in the process. Followed by the separator '#' (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier.*

4.2.2.1.4 Windows 95 & Windows NT

Due to the restrictions imposed by the Windows 95 and Windows NT operating system environments, on the *FileIdentifier* associated with a file the following methodology shall be employed to handle *FileIdentifier(s)* under the above-mentioned operating system environment:

1. *FileIdentifier* Lookup: Upon request for a *"lookup"* of a *FileIdentifier*, a case-sensitive comparison may be performed. If the case-sensitive comparison is not done or if it fails, a case-insensitive comparison shall be performed.
2. Validate *FileIdentifier*: If the *FileIdentifier* is a valid file identifier for Windows 95 or Windows NT then do not apply the following steps.
3. Invalid Characters: A *FileIdentifier* that contains characters considered invalid within a file name of the supported operating system, or not displayable in the current environment shall have them translated into "_" (#005F). Multiple sequential invalid or non-displayable characters shall be translated into a single "_" (#005F) character. Reference appendix 6.7.2 on invalid characters for a complete list.
4. Trailing Periods and Spaces: All trailing "." (#002E) and " " (#0020) shall be removed.
5. FileIdentifier CRC: Since through the above process character information from the original *FileIdentifier* is lost the chance of creating a duplicate *FileIdentifier* in the same directory increases. To greatly reduce the chance of having a duplicate *FileIdentifier* the *file name* shall be modified to contain a CRC of the original *FileIdentifier*.

   If there is a *file extension* then the new *FileIdentifier* shall be composed of up to the first (255 – (length of (new *file extension*) + 1 (for the '.')) – 5 (for the #CRC)) characters constituting the *file name* at this step in the process, followed by the separator '#' (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*, followed by '.' (#002E) and the *file extension* at this step in the process.

Otherwise if there is no *file extension* the new *FileIdentifier* shall be composed of up to the first (255 – 5 (for the #CRC)) characters constituting the *file name* at this step in the process. Followed by the separator '#' (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*.

4.2.2.1.5 UNIX

Due to the restrictions imposed by UNIX operating system environments, on the *FileIdentifier* associated with a file the following methodology shall be employed to handle *FileIdentifier(s)* under the above-mentioned operating system environment:

1. *FileIdentifier* Lookup: Upon request for a *"lookup"* of a *FileIdentifier*, a case-sensitive comparison shall be performed.

2. Validate *FileIdentifier*: If the *FileIdentifier* is a valid UNIX file identifier for the current system environment then do not apply the following steps.

3. Invalid Characters: A *FileIdentifier* that contains characters considered invalid within a UNIX file name for the current system environment, or not displayable in the current environment shall have them translated into "_" (#005E). Multiple sequential invalid or non-displayable characters shall be translated into a single "_" (#005E) character. Reference appendix 6.7.2 on invalid characters for a complete list

4. Long FileIdentifier: In the event that the number of characters constituting the *FileIdentifier* at this step in the process is greater than *MAXNameLength (*maximum name length for the specific UNIX operating system), the new *FileIdentifier* will consist of the first *MAXNameLength-5* characters of the *FileIdentifier* at this step in the process.

5. FileIdentifier CRC: Since through the above process character information from the original *FileIdentifier* is lost the chance of creating a duplicate *FileIdentifier* in the same directory increases. To greatly reduce the chance of having a duplicate *FileIdentifier* the *file name* shall be modified to contain a CRC of the original *FileIdentifier*.

   If there is a *file extension* then the new *FileIdentifier* shall be composed of up to the first (*MAXNameLength* – (length of (new *file extension*) + 1 (for the '.')) – 5 (for the #CRC)) characters constituting the *file name* at this step in the process, followed by the separator '#' (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*, followed by '.' (#002E) and the *file extension* at this step in the process.

   Otherwise if there is no *file extension* the new *FileIdentifier* shall be composed of up to the first (*MAXNameLength* – 5 (for the #CRC)) characters constituting the *file name* at this step in the process. Followed by the separator '#' (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier.*

4.2.2.1.6 OS/400

Due to the restrictions imposed by OS/400 operating system environments, on the *FileIdentifier* associated with a file the following methodology shall be employed to handle *FileIdentifier(s)* under the above mentioned operating system environment.

1. *FileIdentifier* Lookup: Upon request for a "lookup" of a *FileIdentifier*, a case-sensitive comparison may be performed. If the case-sensitive comparison is not done or if it fails, a case-insensitive comparision shall be performed.
2. Validate *FileIdentifier*: If the *FileIdentifier* is a valid file identifier for OS/400 then do not apply the following steps.
3. Invalid Characters: A *FileIdentifier* that contains characters considered invalid within an OS/400 file name, or not displayable in the current environment shall have them translated into "_" (#005F). Multiple sequential invalid or non-displayable characters shall be translated into a single "_" (#005F) character.
4. Trailing Spaces: All trailing " "(#0020) shall be removed.
5. FileIdentifier CRC: Since through the above process character information from the original *FileIdentifier* is lost the chance of creating a duplicate *FileIdentifier* in the same directory increases. To greatly reduce the chance of having a duplicate *FileIdentifier* the filename shall be modified to contain a CRC of the original *FileIdentifier*.
   If there is a file extension then the new *FileIdentifier* shall be composed of up to the first (255 – (length of (new file extension) + 1 (for the '.')) – 5 (for the #CRC)) characters constituting the file name at this step in the process, followed by the separator "#" (#0023); followed by a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*, followed by "." (#002E) and the file extension at this step in the process.

   Otherwise if there is no file extension the new *FileIdentifier* shall be composed of up to the first (255 – 5 (for the new #CRC)) characters constituting the file name at this step in the process. Followed by the separator "#" (#0023); followed by a 4 digit CS0 hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*.

NOTE: *Invalid characters for OS/400 are only the forward slash "/" (#002F) character. Non-displayable characters for OS/400 are any characters that do not translate to code page 500 (EBCDIC Multilingual).*

5. Informative

5.1 Descriptor Lengths

The following table summarizes the UDF limitations on the lengths of the Descriptors described in ECMA 167.

Descriptor	Length in bytes
Anchor Volume Descriptor Pointer	512
Volume Descriptor Pointer	512
Implementation Use Volume Descriptor	512
Primary Volume Descriptor	512
Partition Descriptor	512
Logical Volume Descriptor	no max
Unallocated Space Descriptor	no max
Terminating Descriptor	512
Logical Volume Integrity Descriptor	no max
File Set Descriptor	512
File Identifier Descriptor	Maximum of a Logical Block Size
Allocation Extent Descriptor	24
Indirect Entry	52
Terminal Entry	36
File Entry	Maximum of a Logical Block Size
Extended File Entry	Maximum of a Logical Block Size
Extended Attribute Header Descriptor	24
Unallocated Space Entry	Maximum of a Logical Block Size
Space Bitmap Descriptor	no max
Partition Integrity Entry	N/A
Sparing Table	no max

5.2 Using Implementation Use Areas

5.2.1 Entity Identifiers

Refer to section 2.1.5 on *Entity Identifiers* defined earlier in this document.

5.2.2 Orphan Space

Orphan space may exist within a logical volume, but it is not recommended since some type of logical volume repair facility may reallocate it. Orphan space is defined as space that is not directly or indirectly referenced by any of the non-implementation use descriptors defined in ECMA 167.

NOTE: Any allocated extent for which the only reference resides within an implementation use field is considered orphan space.

5.3 Boot Descriptor

T.B.D.

5.4 Clarification of Unrecorded Sectors

ECMA 167 section 3/8.1.2.2 states

> Any unrecorded constituent sector of a logical sector shall be interpreted as containing all #00 bytes. Within the sector containing the last byte of a logical sector, the interpretation of any bytes after that last byte is not specified by this Part.
>
> A logical sector is unrecorded if the standard for recording allows detection that a sector has been unrecorded and all of the logical sector's constituent sectors are unrecorded. A logical sector should either be completely recorded or unrecorded.

For the purposes of interchange, UDF must clarify the correct interpretation of this section.

This part specifies that an unrecorded sector logically contains #00 bytes. However, the converse argument that a sector containing only #00 bytes is unrecorded is not implied, and such a sector is not an "unrecorded" sector for the purposes of ECMA 167. Only the standard governing the recording of sectors on the media can provide the rule for determining if a sector is unrecorded. For example, a blank check condition would provide correct determination for a WORM device.

The following additional ECMA 167 sections reference the rule defined 3/8.1.2.2: 3/8.4.2, 3/8.8.2, 4/3.1, 4/8.3.1 and 4/8.10. By derivation, paragraph 6.6 (ICB Strategy Type 4096) is also affected. Since unrecorded sectors/blocks are terminating conditions for sequences of descriptors, an implementation must be careful to know that the underlying storage media provides a notion of unrecorded sectors before assuming that not writing to a sector is detectable. Otherwise, reliance on the incorrect converse argument mentioned above may result. Explicit terminating descriptors must be used when an appropriate unrecorded sector would be undetectable.

6. Appendices

6.1 UDF Entity Identifier Definitions

Entity Identifier	Description
"*OSTA UDF Compliant"	Indicates the contents of the specified logical volume or file set is compliant with domain defined by this document.
"*UDF LV Info"	Contains additional Logical Volume identification information.
"*UDF FreeEASpace"	Contains free unused space within the implementation extended attributes space.
"*UDF FreeAppEASpace"	Contains free unused space within the application extended attributes space.
"*UDF DVD CGMS Info"	Contains DVD Copyright Management Information
"*UDF OS/2 EALength"	Contains OS/2 extended attribute length.
"*UDF Mac VolumeInfo"	Contains Macintosh volume information.
"*UDF Mac FinderInfo"	Contains Macintosh finder information.
"*UDF Virtual Partition"	Describes UDF Virtual Partition
"*UDF Sparable Partition"	Describes UDF Sparable Partition
"*UDF OS/400 DirInfo"	OS/400 Extended directory information
"*UDF Sparing Table"	Contains information for handling defective areas on the media
"*UDF Metadata Partition"	Describes UDF Metadata Partition

6.2 UDF Entity Identifier Values

Entity Identifier	Byte Value
"*OSTA UDF Compliant"	#2A, #4F, #53, #54, #41, #20, #55, #44, #46, #20, #43, #6F, #6D, #70, #6C, #69, #61, #6E, #74
“*UDF LV Info”	#2A, #55, #44, #46, #20, #4C, #56, #20, #49, #6E, #66, #6F
"*UDF FreeEASpace"	#2A, #55, #44, #46, #20, #46, #72, #65, #65, #45, #41, #53, #70, #61, #63, #65
"*UDF FreeAppEASpace"	#2A, #55, #44, #46, #20, #46, #72, #65, #65, #41, #70, #70, #45, #41, #53, #70, #61, #63, #65
“*UDF DVD CGMS Info”	#2A, #55, #44, #46, #20, #44, #56, #44, #20, #43, #47, #4D, #53, #20, #49, #6E, #66, #6F
“*UDF OS/2 EALength”	#2A, #55, #44, #46, #20, #4F, #53, #2F, #32, #20, #45, #41, #4C, #65, #6E, #67, #74, #68
“*UDF OS/400 DirInfo”	#2A, #55, #44, #46, #20, #4F, #53, #2F, #34, #30, #30, #20, #44, #69, #72, #49, #6E, #66, #6F
"*UDF Mac VolumeInfo"	#2A, #55, #44, #46, #20, #4D, #61, #63, #20, #56, #6F, #6C, #75, #6D, #65, #49, #6E, #66, #6F
"*UDF Mac FinderInfo"	#2A, #55, #44, #46, #20, #4D, #61, #63, #20, #49, #69, #6E, #64, #65, #72, #49, #6E, #66, #6F
“*UDF Virtual Partition”	#2A, #55, #44, #46, #20, #56, #69, #72, #74, #75, #61, #6C, #20, #50, #61, #72, #74, #69, #74, #69, #6F, #6E
“*UDF Sparable Partition”	#2A, #55, #44, #46, #20, #53, #70, #61, #72, #61, #62, #6C, #65, #20, #50, #61, #72, #74, #69, #74, #69, #6F, #6E
“*UDF Sparing Table”	#2A, #55, #44, #46, #20, #53, #70, #61, #72, #69, #6E, #67, #20, #54, #61, #62, #6C, #65
“*UDF Metadata Partition”	#2A, #55, #44, #46, #20, #4D, #65, #74, #61, #64, #61, #74, #61, #20, #50, #61, #72, #74, #69, #74, #69, #6F, #6E

6.3 Operating System Identifiers

The following sections define the current allowable values for the *OS Class* and *OS Identifier* fields in the *Identifier Suffix* of Entity Identifiers, see 2.1.5.3.

For the most up to date list of values for OS Class and OS Identifier please see the most recent UDF specification. On the OSTA web site, information provided by ISVs who have sent a Developer Registration Form to OSTA can be found, see 6.18.

NOTE: If you wish to add to the OS Class and OS Identifier definitions in the next sections, please contact the OSTA UDF Committee Chairman or post your proposal on the OSTA UDF email reflector, see the OSTA address information listed in **POINTS OF CONTACT** on the first page of this document.

6.3.1 OS Class

The *OS Class* field will identify under which class of operating system the specified descriptor was recorded. The valid values for this field are as follows:

Value	Operating System Class
0	Undefined
1	DOS
2	OS/2
3	Macintosh OS
4	UNIX
5	Windows 9x
6	Windows NT
7	OS/400
8	BeOS
9	Windows CE
10-255	Reserved

6.3.2 OS Identifier

The *OS Identifier* field will identify under which operating system the specified descriptor was recorded. The valid values for this field are as follows:

OS Class	OS Identifier	Operating System Identified
0	Any Value	Undefined
1	0	DOS/Windows 3.x
2	0	OS/2
3	0	Macintosh OS 9 and older.
3	1	Macintosh OS X and later releases.
4	0	UNIX - Generic
4	1	UNIX - IBM AIX
4	2	UNIX - SUN OS / Solaris
4	3	UNIX - HP/UX
4	4	UNIX - Silicon Graphics Irix
4	5	UNIX - Linux
4	6	UNIX - MKLinux
4	7	UNIX - FreeBSD
4	8	UNIX - NetBSD
5	0	Windows 9x – generic (includes Windows 98/ME)
6	0	Windows NT – generic (includes Windows 2000,XP,Server 2003, and later releases based on the same code base)
7	0	OS/400
8	0	BeOS - generic
9	0	Windows CE - generic

6.4 OSTA Compressed Unicode Algorithm

```
/***********************************************************************
 * OSTA compliant Unicode compression, uncompression routines.
 * Copyright 1995 Micro Design International, Inc.
 * Written by Jason M. Rinn.
 * Micro Design International gives permission for the free use of the
 * following source code.
 */
#include <stddef.h>

/***********************************************************************
 * The following two typedef's are to remove compiler dependancies.
 * byte needs to be unsigned 8-bit, and unicode_t needs to be
 * unsigned 16-bit.
 */
typedef unsigned short unicode_t;
typedef unsigned char byte;

/***********************************************************************
 * Takes an OSTA CS0 compressed unicode name, and converts
 * it to Unicode.
 * The Unicode output will be in the byte order
 * that the local compiler uses for 16-bit values.
 * NOTE: This routine only performs error checking on the compID.
 * It is up to the user to ensure that the unicode buffer is large
 * enough, and that the compressed unicode name is correct.
 *
 * RETURN VALUE
 *
 *    The number of unicode characters which were uncompressed.
 *    A -1 is returned if the compression ID is invalid.
 */
int UncompressUnicode(
int numberOfBytes,   /* (Input) number of bytes read from media.  */
byte *UDFCompressed, /* (Input) bytes read from media.            */
unicode_t *unicode)    /* (Output) uncompressed unicode characters. */
{
   unsigned int compID;
   int returnValue, unicodeIndex, byteIndex;

   /* Use UDFCompressed to store current byte being read. */
   compID = UDFCompressed[0];

   /* First check for valid compID. */
   if (compID != 8 && compID != 16)
   {
      returnValue = -1;
   }
   else
   {
      unicodeIndex = 0;
      byteIndex = 1;

      /* Loop through all the bytes. */
      while (byteIndex < numberOfBytes)
      {
         if (compID == 16)
         {
          /*Move the first byte to the high bits of the unicode char. */
            unicode[unicodeIndex] = UDFCompressed[byteIndex++] << 8;
         }
         else
         {
            unicode[unicodeIndex] = 0;
         }
         if (byteIndex < numberOfBytes)
         {
            /*Then the next byte to the low bits. */
            unicode[unicodeIndex] |= UDFCompressed[byteIndex++];
         }
```

```
         unicodeIndex++;
      }
      returnValue = unicodeIndex;
   }
   return(returnValue);
}


/***********************************************************************
 * DESCRIPTION:
 * Takes a string of unicode wide characters and returns an OSTA CS0
 * compressed unicode string. The unicode MUST be in the byte order of
 * the compiler in order to obtain correct results.  Returns an error
 * if the compression ID is invalid.
 *
 * NOTE: This routine assumes the implementation already knows, by
 * the local environment, how many bits are appropriate and
 * therefore does no checking to test if the input characters fit
 * into that number of bits or not.
 *
 * RETURN VALUE
 *
 *    The total number of bytes in the compressed OSTA CS0 string,
 *    including the compression ID.
 *    A -1 is returned if the compression ID is invalid.
 */
int CompressUnicode(
int numberOfChars,   /* (Input) number of unicode characters.   */
int compID,          /* (Input) compression ID to be used.      */
unicode_t *unicode,    /* (Input) unicode characters to compress. */
byte *UDFCompressed) /* (Output) compressed string, as bytes.   */
{
   int byteIndex, unicodeIndex;

   if (compID != 8 && compID != 16)
   {
      byteIndex = -1;  /* Unsupported compression ID ! */
   }
   else
   {
      /* Place compression code in first byte. */
      UDFCompressed[0] = compID;

      byteIndex = 1;
      unicodeIndex = 0;
      while (unicodeIndex < numberOfChars)
      {
         if (compID == 16)
         {
            /* First, place the high bits of the char
             * into the byte stream.
             */
            UDFCompressed[byteIndex++] =
                          (unicode[unicodeIndex] & 0xFF00) >> 8;
         }
         /*Then place the low bits into the stream. */
         UDFCompressed[byteIndex++] = unicode[unicodeIndex] & 0x00FF;
         unicodeIndex++;
      }
   }

   return(byteIndex);
}
```

6.5 CRC Calculation

The following C program may be used to calculate the CRC-CCITT checksum used in the TAG descriptors of ECMA 167.

```
/*
 *     CRC 010041
 */
static unsigned short crc_table[256] = {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
    0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52B5, 0x4294, 0x72F7, 0x62D6,
    0x9339, 0x8318, 0xB37B, 0xA35A, 0xD3BD, 0xC39C, 0xF3FF, 0xE3DE,
    0x2462, 0x3443, 0x0420, 0x1401, 0x64E6, 0x74C7, 0x44A4, 0x5485,
    0xA56A, 0xB54B, 0x8528, 0x9509, 0xE5EE, 0xF5CF, 0xC5AC, 0xD58D,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76D7, 0x66F6, 0x5695, 0x46B4,
    0xB75B, 0xA77A, 0x9719, 0x8738, 0xF7DF, 0xE7FE, 0xD79D, 0xC7BC,
    0x48C4, 0x58E5, 0x6886, 0x78A7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xC9CC, 0xD9ED, 0xE98E, 0xF9AF, 0x8948, 0x9969, 0xA90A, 0xB92B,
    0x5AF5, 0x4AD4, 0x7AB7, 0x6A96, 0x1A71, 0x0A50, 0x3A33, 0x2A12,
    0xDBFD, 0xCBDC, 0xFBBF, 0xEB9E, 0x9B79, 0x8B58, 0xBB3B, 0xAB1A,
    0x6CA6, 0x7C87, 0x4CE4, 0x5CC5, 0x2C22, 0x3C03, 0x0C60, 0x1C41,
    0xEDAE, 0xFD8F, 0xCDEC, 0xDDCD, 0xAD2A, 0xBD0B, 0x8D68, 0x9D49,
    0x7E97, 0x6EB6, 0x5ED5, 0x4EF4, 0x3E13, 0x2E32, 0x1E51, 0x0E70,
    0xFF9F, 0xEFBE, 0xDFDD, 0xCFFC, 0xBF1B, 0xAF3A, 0x9F59, 0x8F78,
    0x9188, 0x81A9, 0xB1CA, 0xA1EB, 0xD10C, 0xC12D, 0xF14E, 0xE16F,
    0x1080, 0x00A1, 0x30C2, 0x20E3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83B9, 0x9398, 0xA3FB, 0xB3DA, 0xC33D, 0xD31C, 0xE37F, 0xF35E,
    0x02B1, 0x1290, 0x22F3, 0x32D2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xB5EA, 0xA5CB, 0x95A8, 0x8589, 0xF56E, 0xE54F, 0xD52C, 0xC50D,
    0x34E2, 0x24C3, 0x14A0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xA7DB, 0xB7FA, 0x8799, 0x97B8, 0xE75F, 0xF77E, 0xC71D, 0xD73C,
    0x26D3, 0x36F2, 0x0691, 0x16B0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xD94C, 0xC96D, 0xF90E, 0xE92F, 0x99C8, 0x89E9, 0xB98A, 0xA9AB,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18C0, 0x08E1, 0x3882, 0x28A3,
    0xCB7D, 0xDB5C, 0xEB3F, 0xFB1E, 0x8BF9, 0x9BD8, 0xABBB, 0xBB9A,
    0x4A75, 0x5A54, 0x6A37, 0x7A16, 0x0AF1, 0x1AD0, 0x2AB3, 0x3A92,
    0xFD2E, 0xED0F, 0xDD6C, 0xCD4D, 0xBDAA, 0xAD8B, 0x9DE8, 0x8DC9,
    0x7C26, 0x6C07, 0x5C64, 0x4C45, 0x3CA2, 0x2C83, 0x1CE0, 0x0CC1,
    0xEF1F, 0xFF3E, 0xCF5D, 0xDF7C, 0xAF9B, 0xBFBA, 0x8FD9, 0x9FF8,
    0x6E17, 0x7E36, 0x4E55, 0x5E74, 0x2E93, 0x3EB2, 0x0ED1, 0x1EF0
};

unsigned short
cksum(s, n)
       register unsigned char *s;
       register int n;
{
       register unsigned short crc=0;

       while (n-- > 0)
          crc = crc_table[(crc>>8 ^ *s++) & 0xff] ^ (crc<<8);

       return crc;
}

/* UNICODE Checksum */
unsigned short
unicode_cksum(s, n)
       register unsigned short *s;
       register int n;
{
       register unsigned short crc=0;
       while (n-- > 0) {
/* Take high order byte first--corresponds to a big endian byte stream. */
              crc = crc_table[(crc>>8 ^ (*s>>8) & 0xff] ^ (crc<<8);
              crc = crc_table[(crc>>8 ^ (*s++ & 0xff)) & 0xff] ^ (crc<<8);
       }
       return crc;
```

```
}

#ifdef MAIN
unsigned char bytes[] = { 0x70, 0x6A, 0x77 };

main()
{
	unsigned short x;

	x = cksum(bytes, sizeof bytes);
	printf("checksum: calculated=%4.4x, correct=%4.4x\en", x, 0x3299);
	exit(0);
}
#endif
```

The CRC table in the previous listing was generated by the following program:

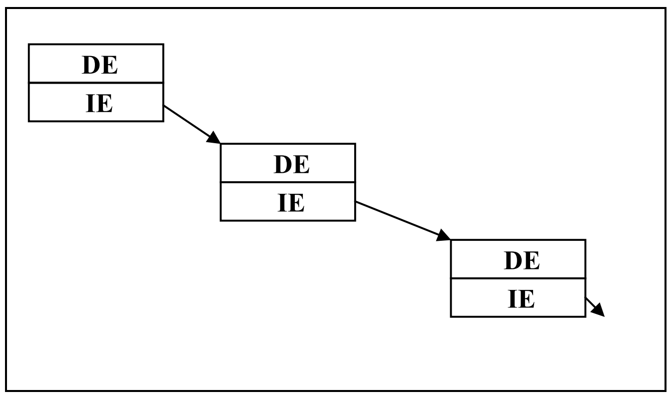
```
#include	<stdio.h>

/*
 *	a.out 010041 for CRC-CCITT
 */

main(argc, argv)
	int argc; char *argv[];
{
	unsigned long crc, poly;
	int n, i;

	sscanf(argv[1], "%lo", &poly);
	if(poly & 0xffff0000){
		fprintf(stderr, "polynomial is too large\en");
		exit(1);
	}

	printf("/*\en *	CRC 0%o\en */\en", poly);
	printf("static unsigned short crc_table[256] = {\en");
	for(n = 0; n < 256; n++){
		if(n % 8 == 0)
			printf("    ");
		crc = n << 8;
		for(i = 0; i < 8; i++){
			if(crc & 0x8000)
				crc = (crc << 1) ^ poly;
			else
				crc <<= 1;
			crc &= 0xFFFF;
		}
		if(n == 255)
			printf("0x%04X ", crc);
		else
			printf("0x%04X, ", crc);
		if(n % 8 == 7)
			printf("\en");
	}
	printf("};\en");
	exit(0);
}
```

All the above CRC code was devised by Don P. Mitchell of AT&T Bell Laboratories and Ned W. Rhodes of Software Systems Group.
It has been published in "Design and Validation of Computer Protocols,"
Prentice Hall, Englewood Cliffs, NJ, 1991, Chapter 3, ISBN 0-13-539925-4.

6.6 Algorithm for ICB Strategy Type 4096

This section describes a strategy for constructing an ICB hierarchy. For ICB Strategy Type 4096 the root ICB hierarchy shall contain 1 direct entry and 1 indirect entry. To indicate that there is 1 direct entry a 1 shall be recorded as a Uint16 in the *StrategyParameter* field of the ICB Tag field. A value of 2 shall be recorded in the *MaximumNumberOfEntries* field of the ICB Tag field.

The indirect entry shall specify the address of another ICB which shall also contain 1 direct entry and 1 indirect entry, where the indirect entry specifies the address of another ICB of the same type. See the figure below:



NOTE: This strategy builds an ICB hierarchy that is a simple linked list of direct entries.

6.7 Identifier Translation Algorithms

The following sample source code examples implement the File Identifier translation algorithms described in this document.

The following basic algorithms may also be used to handle OS specific translations of the *VolumeIdentifier*, *VolumeSetIdentifier*, *LogicalVolumeIdentifier and FileSetIdentifier*.

6.7.1 DOS Algorithm

```
/* OSTA UDF compliant file name translation routine for DOS and     */
/* Windows short namespaces.                                        */
/* Define constants for namespace translation                       */
#define DOS_NAME_LEN 8
#define DOS_EXT_LEN 3
#define DOS_LABEL_LEN 11
#define DOS_CRC_LEN 4
#define DOS_CRC_MODULUS 41

/* Define standard types used in example code.                      */
typedef BOOLEAN int;
typedef short INT16;
typedef unsigned short UINT16;
typedef UINT16 UNICODE_CHAR;
#define FALSE 0
#define TRUE 1
static char crcChar[] =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ#_~-@";

/* FUNCTION PROTOTYPES */
UNICODE_CHAR UnicodeToUpper(UNICODE_CHAR value);
BOOLEAN IsFileNameCharLegal(UNICODE_CHAR value);
BOOLEAN IsVolumeLabelCharLegal(UNICODE_CHAR value);
INT16 NativeCharLength(UNICODE_CHAR value);
BOOLEAN IsDeviceName(UNICODE_CHAR* name, UINT16 nameLen);

/*****************************************************************/
/* UDFDOSName()                                                     */
/* Translate udfName to dosName using OSTA compliant algorithm.     */
/* dosName must be a Unicode string buffer at least 12 characters   */
/* in length.                                                       */
/*****************************************************************/
UINT16 UDFDOSName(UNICODE_CHAR* dosName, UNICODE_CHAR* udfName,
UINT16 udfNameLen)
{
         INT16 index;
         INT16 targetIndex;
         INT16 crcIndex;
         INT16 extLen;
         INT16 nameLen;
         INT16 charLen;
         INT16 overlayBytes;
         INT16 bytesLeft;
         UNICODE_CHAR current;
         BOOLEAN needsCRC;
         UNICODE_CHAR ext[DOS_EXT_LEN];

         needsCRC = FALSE;

         /* Start at the end of the UDF file name and scan for a period    */
         /* ('.'). This will be where the DOS extension starts (if         */
         /* any). */
         index = udfNameLen;
         while (index-- > 0) {
              if (udfName[index] == '.')
                   break;
         }

         if (index < 0) {
            /* There name was scanned to the beginning of the buffer */
            /* and no extension was found. */
            extLen = 0;
```

```
      nameLen = udfNameLen;
   }
   else {
      /* A DOS extension was found, process it first. */
      extLen = udfNameLen - index - 1;
      nameLen = index;
      targetIndex = 0;
      bytesLeft = DOS_EXT_LEN;

      while (++index < udfNameLen && bytesLeft > 0) {
         /* Get the current character and convert it to upper */
         /* case.                                              */
         current = UnicodeToUpper(udfName[index]);
         if (current == ' ') {
            /* If a space is found, a CRC must be appended to */
            /* the mangled file name. */
            needsCRC = TRUE;
         }
         else {
            /* Determine if this is a valid file name char and   */
            /* calculate its corresponding BCS character byte    */
            /* length (zero if the char is not legal or          */
            /* undisplayable on this system).                    */
            charLen = (IsFileNameCharLegal(current)) ?
                         NativeCharLength(current) : 0;

            /* If the char is larger than the available space    */
            /* in the buffer, pretend it is undisplayable.       */
            if (charLen > bytesLeft)
               charLen = 0;

            if (charLen == 0) {
               /* Undisplayable or illegal characters are */
               /* substituted with an underscore ("_"), and */
               /* required a CRC code appended to the mangled */
               /* file name. */
               needsCRC = TRUE;
               charLen = 1;
               current = '_';

               /* Skip over any following undiplayable or */
               /* illegal chars. */
               while (index +1 <udfNameLen &&
                       (!IsFileNameCharLegal(udfName[index + 1]) ||
                            NativeCharLength(udfName[index + 1]) == 0))
                       index++;
            }
            /* Assign the resulting char to the next index in */
            /* the extension buffer and determine how many BCS */
            /* bytes are left. */
            ext[targetIndex++] = current;
            bytesLeft -= charLen;
         }
      }

      /* Save the number of Unicode characters in the extension */
      extLen = targetIndex;

      /* If the extension was too large, or it was zero length */
      /* (i.e. the name ended in a period), a CRC code must be */
      /* appended to the mangled name. */
      if (index < udfNameLen || extLen == 0)
          needsCRC = TRUE;
   }

   /* Now process the actual file name. */
   index = 0;
   targetIndex = 0;
   crcIndex = 0;
   overlayBytes = -1;
   bytesLeft = DOS_NAME_LEN;
   while (index < nameLen && bytesLeft > 0) {
      /* Get the current character and convert it to upper case. */
      current = UnicodeToUpper(udfName[index]);
      if (current ==' ' ||current == '.') {
         /* Spaces and periods are just skipped, a CRC code */
         /* must be added to the mangled file name. */
         needsCRC = TRUE;
      }
      else {
```

```
            /* Determine if this is a valid file name char and */
            /* calculate its corresponding BCS character byte */
            /* length (zero if the char is not legal or */
            /* undisplayable on this system). */
            charLen = (IsFileNameCharLegal(current)) ?
            NativeCharLength(current) : 0;

            /* If the char is larger than the available space in */
            /* the buffer, pretend it is undisplayable. */
            if (charLen > bytesLeft)
                 charLen = 0;

            if (charLen == 0) {
                 /* Undisplayable or illegal characters are */
                 /* substituted with an underscore ("_"), and */
                 /* required a CRC code appended to the mangled */
                 /* file name. */
                 needsCRC = TRUE;
                 charLen = 1;
                 current = '_';

                 /* Skip over any following undiplayable or illegal */
                 /* chars. */
                 while (index +1 <nameLen &&
                           (!IsFileNameCharLegal(udfName[index + 1]) ||
                                NativeCharLength(udfName[index + 1]) == 0))
                           index++;

                 /* Terminate loop if at the end of the file name. */
                 if (index >= nameLen)
                      break;
            }

            /* Assign the resulting char to the next index in the */
            /* file name buffer and determine how many BCS bytes */
            /* are left. */
            dosName[targetIndex++] = current;
            bytesLeft -= charLen;

            /* This figures out where the CRC code needs to start */
            /* in the file name buffer. */
            if (bytesLeft >= DOS_CRC_LEN) {
                 /* If there is enough space left, just tack it */
                 /* onto the end. */
                 crcIndex = targetIndex;
            }
            else {
                 /* If there is not enough space left, the CRC */
                 /* must overlay a character already in the file */
                 /* name buffer. Once this condition has been */
                 /* met, the value will not change. */

                 if (overlayBytes < 0) {
                      /* Determine the index and save the length of */
                      /* the BCS character that is overlayed. It */
                      /* is possible that the CRC might overlay */
                      /* half of a two-byte BCS character depending */
                      /* upon how the character boundaries line up. */
                      overlayBytes = (bytesLeft + charLen > DOS_CRC_LEN)?1 :0;
                      crcIndex = targetIndex - 1;
                 }
            }
       }

       /* Advance to the next character. */
       index++;
  }

  /* If the scan did not reach the end of the file name, or the */
  /* length of the file name is zero, a CRC code is needed. */
  if (index < nameLen || index == 0)
       needsCRC = TRUE;

  /* If the name has illegal characters or and extension, it */
  /* is not a DOS device name. */
  if (needsCRC == FALSE && extLen == 0) {
       /* If this is the name of a DOS device, a CRC code should */
       /* be appended to the file name. */
       if (IsDeviceName(udfName, udfNameLen))
            needsCRC = TRUE;
  }
```

```
    /* Append the CRC code to the file name, if needed. */
    if (needsCRC) {
        /* Get the CRC value for the original Unicode string */
        UINT16 udfCRCValue = CalculateCRC(udfName, udfNameLen);

        /* Determine the character index where the CRC should */
        /* begin. */
        targetIndex = crcIndex;

        /* If the character being overlayed is a two-byte BCS */
        /* character, replace the first byte with an underscore. */
        if (overlayBytes > 0)
            dosName[targetIndex++] = '_';

        /* Append the encoded CRC value with delimiter. */
        dosName[targetIndex++] = '#';
        dosName[targetIndex++] =
            crcChar[udfCRCValue / (DOS_CRC_MODULUS * DOS_CRC_MODULUS)];
        udfCRCValue %= DOS_CRC_MODULUS * DOS_CRC_MODULUS;
        dosName[targetIndex++] =
        crcChar[udfCRCValue / DOS_CRC_MODULUS];
        udfCRCValue %= DOS_CRC_MODULUS;
        dosName[targetIndex++] = crcChar[udfCRCValue];
    }

    /* Append the extension, if any. */
    if (extLen > 0) {
        /* Tack on a period and each successive byte in the */
        /* extension buffer. */
        dosName[targetIndex++] = '.';

        for (index = 0; index < extLen; index++)
            dosName[targetIndex++] = ext[index];
    }

    /* Return the length of the resulting Unicode string. */
    return (UINT16)targetIndex;
}


/*******************************************************************/
/* UDFDOSVolumeLabel() */
/* Translate udfLabel to dosLabel using OSTA compliant algorithm. */
/* dosLabel must be a Unicode string buffer at least 11 characters */
/* in length. */
/*******************************************************************/
UINT16 UDFDOSVolumeLabel(UNICODE_CHAR* dosLabel, UNICODE_CHAR*
udfLabel, UINT16 udfLabelLen)
{
        INT16 index;
        INT16 targetIndex;
        INT16 crcIndex;
        INT16 charLen;
        INT16 overlayBytes;
        INT16 bytesLeft;
        UNICODE_CHAR current;
        BOOLEAN needsCRC;
        needsCRC = FALSE;

        /* Scan end of label to see if there are any trailing spaces. */
        index = udfLabelLen;
        while (index-- > 0) {
                if (udfLabel[index] != ' ')
                break;
        }

        /* If there are trailing spaces, adjust the length of the */
        /* string to exclude them and indicate that a CRC code is */
        /* needed. */
        if (index +1 !=udfLabelLen) {
                udfLabelLen = index + 1;
                needsCRC = TRUE;
        }

        index = 0;
        targetIndex = 0;
        crcIndex = 0;
        overlayBytes = -1;
        bytesLeft = DOS_LABEL_LEN;
        while (index < udfLabelLen && bytesLeft > 0) {
```

```
        /* Get the current character and convert it to upper case. */
        current = UnicodeToUpper(udfLabel[index]);
        if (current == '.') {
                /* Periods are just skipped, a CRC code must be added */
                /* to the mangled file name. */
                needsCRC = TRUE;
        }
        else {
                /* Determine if this is a valid file name char and */
                /* calculate its corresponding BCS character byte */
                /* length (zero if the char is not legal or */
                /* undisplayable on this system). */
                charLen = (IsVolumeLabelCharLegal(current)) ?
                NativeCharLength(current) : 0;

                /* If the char is larger than the available space in */
                /* the buffer, pretend it is undisplayable. */
                if (charLen > bytesLeft)
                charLen = 0;
                if (charLen == 0) {
                        /* Undisplayable or illegal characters are */
                        /* substituted with an underscore ("_"), and */
                        /* required a CRC code appended to the mangled */
                        /* file name. */
                        needsCRC = TRUE;
                        charLen = 1;
                        current = '_';

                        /* Skip over any following undiplayable or illegal */
                        /* chars. */
                        while (index +1 <udfLabelLen &&
                                (!IsVolumeLabelCharLegal(udfLabel[index + 1]) ||
                                NativeCharLength(udfLabel[index + 1]) == 0))
                                        index++;

                        /* Terminate loop if at the end of the file name. */
                        if (index >= udfLabelLen)
                                break;
                }

                /* Assign the resulting char to the next index in the */
                /* file name buffer and determine how many BCS bytes */
                /* are left. */
                dosLabel[targetIndex++] = current;
                bytesLeft -= charLen;

                /* This figures out where the CRC code needs to start */
                /* in the file name buffer. */
                if (bytesLeft >= DOS_CRC_LEN) {
                        /* If there is enough space left, just tack it */
                        /* onto the end. */
                        crcIndex = targetIndex;
                }
                else {
                        /* If there is not enough space left, the CRC */
                        /* must overlay a character already in the file */
                        /* name buffer. Once this condition has been */
                        /* met, the value will not change. */
                        if (overlayBytes < 0) {
                                /* Determine the index and save the length of */
                                /* the BCS character that is overlayed. It */
                                /* is possible that the CRC might overlay */
                                /* half of a two-byte BCS character depending */
                                /* upon how the character boundaries line up. */
                                overlayBytes = (bytesLeft + charLen > DOS_CRC_LEN)
                                ?1 :0;
                                crcIndex = targetIndex - 1;
                        }
                }
        }

        /* Advance to the next character. */
        index++;
}

/* If the scan did not reach the end of the file name, or the */
/* length of the file name is zero, a CRC code is needed. */
if (index < udfLabelLen || index == 0)
        needsCRC = TRUE;

/* Append the CRC code to the file name, if needed. */
```

```
	if (needsCRC) {
		/* Get the CRC value for the original Unicode string */
		UINT16 udfCRCValue = CalculateCRC(udfName, udfNameLen);

		/* Determine the character index where the CRC should */
		/* begin. */
		targetIndex = crcIndex;

		/* If the character being overlayed is a two-byte BCS */
		/* character, replace the first byte with an underscore. */
		if (overlayBytes > 0)
			dosLabel[targetIndex++] = '_';

		/* Append the encoded CRC value with delimiter. */
		dosLabel[targetIndex++] = '#';
		dosLabel[targetIndex++] =
		crcChar[udfCRCValue / (DOS_CRC_MODULUS * DOS_CRC_MODULUS)];
		udfCRCValue %= DOS_CRC_MODULUS * DOS_CRC_MODULUS;
		dosLabel[targetIndex++] =
		crcChar[udfCRCValue / DOS_CRC_MODULUS];
		udfCRCValue %= DOS_CRC_MODULUS;
		dosLabel[targetIndex++] = crcChar[udfCRCValue];
	}

	/* Return the length of the resulting Unicode string. */
	return (UINT16)targetIndex;
}


/*****************************************************************/
/* UnicodeToUpper()    */
/* Convert the given character to upper-case Unicode. */
/*****************************************************************/
UNICODE_CHAR UnicodeToUpper(UNICODE_CHAR value)
{
	/* Actual implementation will vary to accommodate the target */
	/* operating system API services. */
	/* Just handle the ASCII range for the time being. */
	return (value >= 'a' && value <= 'z') ?
		value - ('a' - 'A') : value;
}


/*****************************************************************/
/* IsFileNameCharLegal() */
/* Determine if this is a legal file name id character. */
/*****************************************************************/
BOOLEAN IsFileNameCharLegal(UNICODE_CHAR value)
{
	/* Control characters are illegal. */
	if (value <' ')
		return FALSE;

	/* Test for illegal ASCII characters. */
	switch (value) {
		case '\\':
		case '/':
		case ':':
		case '*':
		case '?':
		case '\"':
		case '<':
		case '>':
		case '|':
		case ';':
		case '^':
		case ',':
		case '&':
		case '+':
		case '=':
		case '[':
		case ']':
			return FALSE;

		default:
			return TRUE;
	}
}


/*****************************************************************/
```

```
/* IsVolumeLabelCharLegal() */
/* Determine if this is a legal volume label character. */
/****************************************************************/
BOOLEAN IsVolumeLabelCharLegal(UNICODE_CHAR value)
{
	/* Control characters are illegal. */
	if (value <' ')
		return FALSE;

	/* Test for illegal ASCII characters. */
	switch (value) {
		case '\\':
		case '/':
		case ':':
		case '*':
		case '?':
		case '\"':
		case '<':
		case '>':
		case '|':
		case '.':
		case ';':
		case '^':
		case ',':
		case '&':
		case '+':
		case '=':
		case '[':
		case ']':
			return FALSE;

		default:
			return TRUE;
	}
}


/****************************************************************/
/* NativeCharLength() */
/* Determines the corresponding native length (in bytes) of the */
/* given Unicode character. Returns zero if the character is */
/* undisplayable on the current system. */
/****************************************************************/
INT16 NativeCharLength(UNICODE_CHAR value)
{
	/* Actual implementation will vary to accommodate the target */
	/* operating system API services. */

	/* This is an example of a conservative test. A better test */
	/* will utilize the platform’s language/codeset support to */
	/* determine how wide this character is when converted to the */
	/* active variable width character set. */
	return 1;
}


/****************************************************************/
/* IsDeviceName() */
/* Determine if the given Unicode string corresponds to a DOS */
/* device name (e.g. "LPT1", "COM4", etc.). Since the set of */
/* valid device names with vary from system to system, and */
/* a means for determining them might not be readily available, */
/* this functionality is only suggested as an optional */
/* implementation enhancement. */
/****************************************************************/
BOOLEAN IsDeviceName(UNICODE_CHAR* name, UINT16 nameLen)
{
	/* Actual implementation will vary to accommodate the target */
	/* operating system API services. */
	/* Just return FALSE for the time being. */
	return FALSE;
}
```

6.7.2 OS/2, Macintosh,Windows 95, Windows NT and UNIX Algorithm

```
/***********************************************************************
 * OSTA UDF compliant file name translation routine for OS/2,
 * Windows 95, Windows NT, Macintosh and UNIX.
 * Copyright 1995 Micro Design International, Inc.
 * Written by Jason M. Rinn.
 * Micro Design International gives permission for the free use of the
 * following source code.
 ***********************************************************************
 *
 * To use these routines with different operating systems.
 *
 * OS/2
 *   Define OS2
 *   Define MAXLEN = 254
 *
 * Windows 95
 *   Define WIN_95
 *   Define MAXLEN = 255
 *
 * Windows NT
 *   Define WIN_NT
 *   Define MAXLEN = 255
 *
 * Macintosh:
 *   Define MAC.
 *   Define MAXLEN = 31.
 *
 * UNIX
 *   Define UNIX.
 *   Define MAXLEN as specified by unix version.
 */

#define ILLEGAL_CHAR_MARK 0x005F
#define CRC_MARK          0x0023
#define EXT_SIZE           5
#define TRUE               1
#define FALSE              0
#define PERIOD            0x002E
#define SPACE             0x0020

/***********************************************************************
 * The following two typedef's are to remove compiler dependancies.
 * byte needs to be unsigned 8-bit, and unicode_t needs to
 * be unsigned 16-bit.
 */
typedef unsigned int unicode_t;
typedef unsigned char byte;

/*** PROTOTYPES ***/
int IsIllegal(unicode_t ch);
unsigned short unicode_cksum(register unsigned short *s, register int n);

/* Define a function or macro which determines if a Unicode character is
 * printable under your implementation.
 */
int UnicodeIsPrint(unicode_t);

/***********************************************************************
 * Translates a long file name to one using a MAXLEN and an illegal
 * char set in accord with the OSTA requirements.  Assumes the name has
 * already been translated to Unicode.
 *
 * RETURN VALUE
 *
 *    Number of unicode characters in translated name.
 */
int UDFTransName(
unicode_t *newName,/*(Output)Translated name. Must be of length MAXLEN*/
unicode_t *udfName, /* (Input)  Name from UDF volume.*/
```

```
int udfLen,         /* (Input)  Length of UDF Name.  */
{
   int index, newIndex = 0, needsCRC = FALSE;
   int extIndex, newExtIndex = 0, hasExt = FALSE;
#ifdef (OS2 | WIN_95 | WIN_NT)
   int trailIndex = 0;
#endif
   unsigned short valueCRC;
   unicode_t current;
   const char hexChar[] = "0123456789ABCDEF";

   for (index = 0; index < udfLen; index++)
   {
      current = udfName[index];

      if (IsIllegal(current) || !UnicodeIsPrint(current))
      {
         needsCRC = TRUE;
        /* Replace Illegal and non-displayable chars with underscore. */
         current = ILLEGAL_CHAR_MARK;
         /* Skip any other illegal or non-displayable characters. */
         while(index+1 < udfLen && (IsIllegal(udfName[index+1])
                     || !UnicodeIsPrint(udfName[index+1])))
         {
            index++;
         }
      }

      /* Record position of extension, if one is found. */
      if (current == PERIOD && (udfLen - index -1) <= EXT_SIZE)
      {
         if (udfLen == index + 1)
         {
            /* A trailing period is NOT an extension. */
            hasExt = FALSE;
         }
         else
         {
            hasExt = TRUE;
            extIndex = index;
            newExtIndex = newIndex;
         }
      }

#ifdef (OS2 | WIN_95 | WIN_NT)
      /* Record position of last char which is NOT period or space. */
      else if (current != PERIOD && current != SPACE)
      {
         trailIndex = newIndex;
      }
#endif

      if (newIndex < MAXLEN)
      {
         newName[newIndex++] = current;
      }
      else
      {
         needsCRC = TRUE;
      }
   }

#ifdef (OS2 | WIN_95 | WIN_NT)
   /* For OS2, 95 & NT, truncate any trailing periods and\or spaces. */
   if (trailIndex != newIndex - 1)
   {
      newIndex = trailIndex + 1;
      needsCRC = TRUE;
      hasExt = FALSE; /* Trailing period does not make an extension. */
   }
#endif
```

```
   if (needsCRC)
   {
      unicode_t ext[EXT_SIZE];
      int localExtIndex = 0;
      if (hasExt)
      {
         int maxFilenameLen;
         /* Translate extension, and store it in ext. */
         for(index = 0; index<EXT_SIZE && extIndex + index +1 < udfLen;
              index++ )
         {
            current = udfName[extIndex + index + 1];

            if (IsIllegal(current) || !UnicodeIsPrint(current))
            {
               needsCRC = 1;
               /* Replace Illegal and non-displayable chars
                * with underscore.
                */
               current = ILLEGAL_CHAR_MARK;
               /* Skip any other illegal or non-displayable
                * characters.
                */
               while(index + 1 < EXT_SIZE
                           && (IsIllegal(udfName[extIndex + index + 2])
                           || !UnicodeIsPrint(udfName[extIndex + index + 2])))
               {
                  index++;
               }
            }
            ext[localExtIndex++] = current;
         }

         /* Truncate filename to leave room for extension and CRC. */
         maxFilenameLen = ((MAXLEN - 5) - localExtIndex - 1);
         if (newIndex > maxFilenameLen)
         {
            newIndex = maxFilenameLen;
         }
         else
         {
            newIndex = newExtIndex;
         }
      }
      else if (newIndex > MAXLEN - 5)
      {
         /*If no extension, make sure to leave room for CRC. */
         newIndex = MAXLEN - 5;
      }
      newName[newIndex++] = CRC_MARK; /* Add mark for CRC. */

      /*Calculate CRC from original filename from FileIdentifier. */
      valueCRC = unicode_cksum(udfName, udfLen);
      /* Convert 16-bits of CRC to hex characters. */
      newName[newIndex++] = hexChar[(valueCRC & 0xf000) >> 12];
      newName[newIndex++] = hexChar[(valueCRC & 0x0f00) >> 8];
      newName[newIndex++] = hexChar[(valueCRC & 0x00f0) >> 4];
      newName[newIndex++] = hexChar[(valueCRC & 0x000f)];

      /* Place a translated extension at end, if found. */
      if (hasExt)
      {
         newName[newIndex++] = PERIOD;
         for (index = 0;index < localExtIndex ;index++ )
         {
            newName[newIndex++] = ext[index];
         }
      }
   }
   return(newIndex);
}
```

```
#ifdef (OS2 | WIN_95 | WIN_NT)
/***********************************************************************
 * Decides if a Unicode character matches one of a list
 * of ASCII characters.
 * Used by OS2 version of IsIllegal for readability, since all of the
 * illegal characters above 0x0020 are in the ASCII subset of Unicode.
 * Works very similarly to the standard C function strchr().
 *
 * RETURN VALUE
 *
 *    Non-zero if the Unicode character is in the given ASCII string.
 */
int UnicodeInString(
unsigned char *string,  /* (Input) String to search through.   */
unicode_t ch)  /* (Input) Unicode char to search for. */
{
   int found = FALSE;
   while (*string != '\0' && found == FALSE)
   {
      /* These types should compare, since both are unsigned numbers. */
      if (*string == ch)
      {
         found = TRUE;
      }
      string++;
   }
   return(found);
}
#endif /* OS2 */

/***********************************************************************
 * Decides whether the given character is illegal for a given OS.
 *
 * RETURN VALUE
 *
 *    Non-zero if char is illegal.
 */
int IsIllegal(unicode_t ch)
{
#ifdef MAC
   /* Only illegal character on the MAC is the colon. */
   if (ch == 0x003A)
   {
      return(1);
   }
   else
   {
      return(0);
   }
#elif defined UNIX
   /* Illegal UNIX characters are NULL and slash. */
   if (ch == 0x0000 || ch == 0x002F)
   {
      return(1);
   }
   else
   {
      return(0);
   }
#elif defined (OS2 | WIN_95 | WIN_NT)
   /* Illegal char's for OS/2 according to WARP toolkit. */
   if (ch < 0x0020 || UnicodeInString("\\/:*?\"<>|", ch))
   {
      return(1);
   }
   else
   {
      return(0);
   }
#endif
}
```

6.8 Extended Attribute Header Checksum Algorithm

```
/*
 * Calculates a 16-bit checksum of the Implementation Use
 * Extended Attribute header or Application Use Extended Attribute
 * header.  The fields AttributeType through ImplementationIdentifier
 * (or ApplicationIdentifier) inclusively represent the
 * data covered by the checksum (48 bytes).
 *
 */

Uint16  ComputeEAChecksum(byte *data)
{
       Uint16 checksum = 0;
       Uint          count;

       for( count = 0; count < 48; count++)
       {
              checksum += *data++;
       }

       return(checksum );
}
```

6.9 Requirements for DVD-ROM

This appendix defines the requirements and restrictions for UDF formatted DVD-ROM discs.

- DVD-ROM discs shall be mastered with the UDF file system
- DVD-ROM discs shall consist of a single volume and a single partition.

NOTE: The disc may also include the ISO 9660 file system. If the disc contains both UDF and ISO 9660 file systems it shall be known as a *UDF Bridge* disc. This *UDF Bridge* disc will allow playing DVD-ROM media in computers, which may only support ISO 9660. As UDF computer implementations are provided, the need for ISO 9660 will disappear, and future discs should contain only UDF.

If you intend to do any DVD development with UDF, please make sure that you fill out the OSTA UDF Developer Registration Form located in appendix 6.18. For planned operating system, check the *Other* box and write in DVD.

6.9.1 Constraints imposed on UDF by DVD-Video

This section describes the restrictions and requirements for UDF formatted DVD-Video discs for dedicated DVD content players. DVD-Video is one specific application of DVD-ROM using the UDF format for the home consumer market. Due to limited computing resources within a DVD player, restrictions and requirements were created so that a DVD player would not have to support every feature of the UDF specification.

All DVD-Video discs shall be mastered to contain all required data as specified by ECMA 167 (2^{nd} edition) and UDF 1.02. This will ease playing of DVD-Video in computer systems. Examples of such data include the time, date, permission bits, and a Free Space Table (indicating no free space). While DVD player implementations may ignore these fields, a UDF computer system implementation will not. Both entertainment-based and computer-based content can reside on the same disc.

NOTE 1: DVD-Video discs mastered according to a UDF revision other than 1.02 may not be compatible with DVD-Video players. DVD-Video players expect media in UDF 1.02 format.

In an attempt to reduce code size and improve performance, all division described is integer arithmetic; all denominators shall be 2^n, such that all divisions may be carried out via logical shift operations.

- A DVD player shall only support UDF and not ISO 9660.
- Originating systems shall constrain individual files to be less than or equal to 2^{30} - *Logical Block Size* bytes in length.
- The data of each file shall be recorded as a single extent. Each File Entry shall be recorded using the ICB Strategy Type 4.

- File and directory names shall be compressed as 8 bits per character using OSTA Compressed Unicode format.
- A DVD player shall not be required to follow symbolic links to any files.
- The DVD-Video files shall be stored in a subdirectory named "VIDEO_TS" directly under the root directory. Directory names are standardized in the *DVD Specifications for Read-Only Disc* document.

  NOTE 2: The *DVD Specifications for Read-Only Disc* is a document, published by the DVD Format/Logo Licensing Corporation, see 6.9.3. This document describes the names of all DVD-Video files and a DVD-Video directory, which will be stored on the media, and additionally, describes the contents of the DVD-Video files.
- The file named "VIDEO_TS.IFO" in the VIDEO_TS subdirectory shall be read first.

All the above constraints apply only to the directory and files that the DVD player needs to access. There may be other files and directories on the media which are not intended for the DVD player and do not meet the above listed constraints. These other files and directories are ignored by the DVD player. This is what enables the ability to have both entertainment-based and computer-based content on the same disc.

6.9.2 How to read a UDF DVD-Video disc

This section describes the basic procedures that a DVD player would go through to read a UDF formatted DVD-Video disc.

6.9.2.1 Step 1. Volume Recognition Sequence

Find an ECMA 167 Descriptor in a volume recognition area, which shall start at logical sector 16.

6.9.2.2 Step 2. Anchor Volume Descriptor Pointer

The Anchor Volume Descriptor Pointer, which is located at an anchor point, must be found. Duplicate anchor points shall be recorded at logical sector 256 and logical sector N, where N is the highest numbered logical sector on the disc.

A DVD player only needs to look at logical sector 256; the copy at logical sector N is redundant and only needed for defect tolerance. The Anchor Volume Descriptor Pointer contains three things of interest:

1. Static structures that may be used to identify and verify integrity of the disc.
2. Location of the Main Volume Descriptor Sequence (absolute logical sector number)
3. Length of the Main Volume Descriptor Sequence (bytes)

The data located in bytes 0-3 and 5 of the Anchor Volume Descriptor Pointer may be used for format verification if desired. Verifying the Tag Checksum in byte 4 and

Descriptor CRC in bytes 8-11 are good additional verifications to perform. MVDS_Location and MVDS_Length are read from this structure.

6.9.2.3 Step 3. Volume Descriptor Sequence

Read logical sectors:

MVDS_Location through MVDS_Location + (MVDS_Length - 1) / SectorSize

The logical sector size shall be 2048 bytes for DVD media. If this sequence cannot be read, a Reserve Volume Descriptor Sequence should be read.

The Partition Descriptor shall be a descriptor with a tag identifier of 5. The partition number and partition location shall be recorded in logical sector number.

Partition_Location and Partition_Length are obtained from this structure.

The Logical Volume Descriptor shall be a descriptor with a tag identifier of 6. The location and length of the File Set Descriptor shall be recorded in the Logical Volume Descriptor.

FSD_Location, and FSD_Length are returned from this structure.

6.9.2.4 Step 4. File Set Descriptor

The File Set Descriptor is located at logical sector numbers:

Partition_Location + FSD_Location through
Partition_Location + FSD_Location + (FSD_Length - 1) / BlockSize

RootDir_Location and RootDir_Length shall be read from the File Set Descriptor in logical block number.

6.9.2.5 Step 5. Root Directory File Entry

RootDir_Location and RootDir_Length define the location of a File Entry. The File Entry describes the data space and permissions of the root directory.

The location and length of the Root Directory is returned.

6.9.2.6 Step 6. Root Directory

Parse the data in the root directory extent to find the VIDEO_TS subdirectory.

Find the VIDEO_TS File Identifier Descriptor. The name shall be in 8 bit compressed UDF format. Verify that VIDEO_TS is a directory.

Read the File Identifier Descriptor and find the location and length of a File Entry describing the VIDEO_TS directory.

6.9.2.7 Step 7. File Entry of VIDEO_TS

The File Entry found in the step above describes the data space and permissions of the VIDEO_TS directory.

The location and length of the VIDEO_TS directory is returned.

6.9.2.8 Step 8. VIDEO_TS directory

The extent found in the step above contains sets of File Identifier Descriptors. In this pass, verify that the entry points to a file and is named VIDEO_TS.IFO.

6.9.2.9 Step 9. File Entry of VIDEO_TS.IFO

The File Entry found in the step above describes the data space and permissions of the VIDEO_TS.IFO file.

The location and length of the VIDEO_TS.IFO file is returned.

Further files can be found in the same manner as the VIDEO_TS.IFO file when needed.

6.9.3 Obtaining DVD Documents

To obtain a copy of the *DVD Specifications for Read-Only Disc* document as well as other DVD related material, contact:

DVD Format/Logo Licensing Corporation
Daimon Urbanist Bldg. 6F,
2-3-6 Shibadaimon, Minato-ku,
Tokyo, 105-0012 JAPAN

TEL: +81-3-5777-2883
FAX: +81-3-5777-2884

6.10 Recommendations for CD Media

CD Media (CD-R and CD-RW) requires special consideration due to its nature. CD was originally designed for Read-Only applications, which affects the way in which it is written. The following guidelines are established to ensure interchange.

Each file and directory shall be described by a single direct ICB. The ICB should be written after the file data to allow for data underruns during writing, which will cause logical gaps in the file data. The ICB can be written afterward which will correctly identify all extents of the file data. The ICB shall be written in the data track, the file system track (if it exists), or both.

6.10.1 Use of UDF on CD-R media

For CD-R, the rules of section 6.11 apply with the following additions:

The VAT may be located by using READ TRACK INFORMATION (for unfinished media) or READ TOC or READ CD RECORDED CAPACITY for finished media. See X3T10-1048D (SCSI-3 Multi Media Commands).

6.10.1.1 Mode requirements for CD-R

- Writing shall use Mode 1 or Mode 2 Form 1 sectors. On one disc, either Mode 1 or Mode 2 Form 1 shall be used; a mixture of Mode 1 and Mode 2 Form 1 sectors on one disc is not allowed.
  NOTE: According to the Multisession CD Specification, all data sessions on a disc must be of the same type (Mode 1, or Mode 2 Form 1).
- If Mode 2 Form 1 is used, then the subheader bytes of all sectors used by the user data files and by the UDF structures shall have the following value:

  File number = 0

  Channel number = 0

  Submode = 08h

  Coding information = 0

6.10.1.2 UDF Bridge format for CD-R

If an ISO 9660 bridge disc contains Mode 2 Form 1 sectors, then the CD-ROM XA extensions for ISO 9660 must be used. Further the rules of section 6.11.4 apply.

6.10.2 Use of UDF on CD-RW media

CD-RW media is randomly readable and block writable. This means that while any individual sector may be read, writing must occur in blocks containing multiple sectors. CD-RW systems do not provide for sparing of bad areas. Writing rules and sparing mechanisms have been defined.

6.10.2.1 Requirements

- Writing which conforms to this section of the standard shall be performed using fixed length packets.
- Writing shall be performed using Mode 1 or Mode 2, Form 1 sectors. On one disc, either Mode 1 or Mode 2 Form 1 shall be used.
  NOTE: According to the Multisession CD Specification, all data sessions on a disc must be of the same type (Mode 1, or Mode 2 Form 1).
- If Mode 2 Form 1 is used, then the subheader bytes of all sectors used by the user data files and by the UDF structures shall have the following value:
  File number = 0
  Channel number = 0
  Submode = 08h
  Coding information = 0
- The host shall perform read/modify/write to enable the apparent writing of single 2K sectors.
- The Packet Length shall be set when the disc is formatted. The Packet Length shall be 32 sectors (64 KB).
- Defective packets known at format time shall be allocated by the Non-Allocatable Space Stream (see 3.3.7.2).
- Sparing shall be managed by the host via the sparable partition and a Sparing Table.
- Discs shall be formatted prior to use.

6.10.2.2 Formatting

Formatting shall consist of writing a lead-in, user data area, and lead-out. These areas may be written in any order. A verification pass may follow this physical format. Defective packets found during the verification pass shall be *enumerated* in the *Non-Allocatable Space* Stream (see 3.3.7.2). Finally, file system root structures shall be recorded. These mandatory file system and root structures include the Volume Recognition Sequence, Anchor Volume Descriptor Pointers, a Volume Descriptor Sequence, a File Set Descriptor and a Root Directory.

The Anchor Volume Descriptor Pointers shall be recorded at sectors 256 and N - 256, where N is the Logical Sector Number of the last addressable sector.

Allocation for sparing shall occur during the format process. The sparing allocation may be zero in length.

The free space descriptors shall be recorded and shall reflect space allocated to defective areas and sector sparing areas. The format may include all available space on the medium. However, if requested by the user, a subset may be formatted to save formatting time. That smaller format may be later “grown” to the full available space.

6.10.2.3 Growing the Format

If the medium is partially formatted, it may be later grown to a larger size. This operation consists of:

- Optionally erase the lead-in of the last session.
- Optionally erase the lead-out of the last session.
- Write packets beginning immediately after the last recorded packet.
- Update the Sparing Table to reflect any new spare areas
- Adjust the Partition Map as appropriate
- Update the Unallocated Space Bitmap or Table to show new available area
- Move the last AVDP to the new N - 256
- Write the lead-in (which reflects the new track size)
- Write the lead-out

6.10.2.4 Host Based Defect Management

The host shall perform defect management operations. The CD format was defined without any defect management; to be compatible with existing technology and components, the host must manage defects. There are two levels of defect management: Marking bad sectors at format time and on-line sparing. The host shall keep the tables on the media current.

6.10.2.5 Read Modify Write Operation

CD-RW media requires large writable units, as each unit incurs a 14KB overhead. The file system requires a 2KB writable unit. The difference in write sizes is handled by a read-modify-write operation by the host. An entire packet is read, the appropriate portions are modified, and the entire packet written to the CD.
Note that packets may not be aligned to 32 sector boundaries.

6.10.2.6 Levels of Compliance

6.10.2.6.1 Level 1

The disc shall be formatted with exactly one lead-in, program area, and lead-out. The program area shall contain exactly one track.

6.10.2.6.2 Level 2

The last session shall contain the UDF file system. All prior sessions shall be contained in one read-only partition.

6.10.2.6.3 Level 3

No restrictions shall apply.

6.10.3 Multisession and Mixed Mode

For CD-R and CD-RW, the multisession and bridge disc rules of 6.11 apply with the following additions:

If random write mode is used, the media may be formatted with zero or one audio sessions followed by exactly one writable data session containing one track. Other session configurations are possible but not described here.

When recorded in Random Access mode, a duplicate Volume Recognition Sequence should be recorded beginning at sector *N* - 16.
CD multisession discs may also contain audio sessions. The UDF Bridge format allows CD enhanced discs to be created, see an example in 6.11.5.

6.11 Common aspects of recording for different media

In the following sections, common aspects of recording for different media are described. These aspects are:

- Real-Time files
- Incremental recording using VAT
- Multisession discs
- UDF Bridge discs

Media that do not support sessions are assumed to have a single session that starts at logical sector zero and ends at the highest addressable logical sector number. Media that do not support tracks are assumed to have a single track per session with the same size and start address as the session. For some media different terms may be used for 'track' and 'session', e.g. for DVD+R, a track is called a Fragment.

6.11.1 Real-Time Files

A Real-Time file shall be identified by file type 249 in the File Type field of the file's ICB Tag. A Real-Time file is a file that requires a minimum data-transfer rate when writing or reading, for example, audio and video data. For these files, special read and write commands are needed. For example for CD and DVD devices these special commands can be found in the SCSI-3 MMC command set specification.

6.11.2 Incremental recording using VAT

This type of recording is used on sequential media that have a Virtual Partition Map recorded in the Logical Volume Descriptor, see 2.2.8. VAT usage is described in 2.2.11. The VAT ICB is recorded at the highest recorded Logical Sector Number on the medium. This logical sector number may be located using the READ TRACK INFORMATION command for the relevant medium, e.g. see SCSI-3 Multi Media Commands.
ECMA 167 requires at least two Anchor Volume Descriptor Pointers (AVDP) at Logical Sector Numbers 256, *N* or (*N* - 256), where *N* is the highest valid Logical Sector Number on the medium, see 2.2.3. Because the VAT ICB is recorded as last, *N* cannot be used for an AVDP. Only if the last session is closed, there shall be an AVDP at (*N* - 256).
For open sessions, the file system may be in an intermediate state before closing and still be interchangeable, but not strictly in compliance with ECMA 167. In the intermediate state, only one AVDP exists. It should exist at sector 256 or, if not possible due to a track reservation, it shall exist at sector 512. An AVDP at 512 must be ignored if an AVDP at 256, N-256, or *N* exists. An AVDP at 512 can point to a temporary Volume Descriptor Sequence that is only used in the intermediate state.

Implementations should place file system control structures into virtual space and file data into real space. Reader implementations may cache the entire VAT. The size of the VAT should be considered by any UDF originating software.

6.11.2.1 Requirements

- An intermediate state is allowed for media on which only one AVDP is recorded; this single AVDP shall be at sector 256 or sector 512 and according to the multisession rules in 6.11.3.
- The Logical Volume Integrity Descriptor shall be recorded and the volume marked as open. Logical Volume integrity can be verified by finding the VAT ICB at the last recorded Logical Sector Number. If the VAT ICB is present, the volume is clean; otherwise it is dirty.
- The Partition Header Descriptor shall specify no Unallocated Space Table, no Unallocated Space Bitmap, no Freed Space Table, and no Freed Space Bitmap. The drive is capable of reporting free space directly, eliminating the need for a separate descriptor.
- Each surface shall contain 0 or 1 read-only partitions, 0 or 1 write-once partitions, and 0 or 1 virtual partitions. Media using VAT should contain 1 write-once partition and 1 virtual partition.

6.11.2.2 End of session data

Some Read-Only drives (e.g. CD-ROM, DVD-ROM) can only read closed sessions. The last complete session on the disc shall conform completely to ECMA 167 and have two AVDPs recorded. This shall be accomplished by writing data according to the End of session data table below.

End of session data

Count	Description
1	Anchor Volume Descriptor Pointer
255	Implementation specific. May contain user data, file system structures, and/or link areas.
1	VAT ICB.

The implementation specific data may contain repeated copies of the VAT and VAT ICB. Compatibility with drives that do not accurately report the location of the last sector will be enhanced. Implementations shall ensure that enough space is available to record the end of session data. Recording the end of session data brings a volume into compliance with ECMA 167.

6.11.3 Multisession Usage

The Volume Recognition Sequence and Anchor Volume Descriptor Pointer locations are specified by ECMA 167 to be at a location relative to the beginning of the disc. The beginning of a disc shall be determined from a base address *S* for the purposes of finding the VRS and AVDP.
'*S*' is the logical sector number of the first data sector in the last existent session of the volume. It is the same value used in multisession ISO 9660 recording. The first track in the last session shall be a data track.
'*N*' is the logical sector number of the highest addressable data sector on a volume.

There shall be no more than one writable partition or session at one time, and this session shall be the last session on the disc.

A new Main and Reserve Volume Descriptor Sequence may exist in each added session, and may be different than earlier VDSs.

If the last session on a medium does not contain a valid UDF file system, the disc is not a UDF disc. Only the UDF structures in the last session, and any UDF structures and data referenced through them, are valid.

The UDF session may contain pointers to data or metadata in other sessions, pointers to data or metadata only within the UDF session, or a combination of both.

6.11.3.1 Volume Recognition Sequence

The following descriptions are added to UDF (see also ECMA 167 Part 2) in order to handle a multisession disc.

- The volume recognition area of the UDF Bridge format (see 6.11.4) shall be the part of the volume space starting at sector *S* + 16 (assuming 2K sectors).
- The volume recognition space shall end in the session in which it begins. As a result of this definition, the volume recognition area always exists in the last session of a disc.

6.11.3.2 Anchor Volume Descriptor Pointer

The Anchor Volume Descriptor Pointers (AVDP) shall be recorded on at least 2 of the following logical sector numbers: *S* + 256, *N* - 256 and *N*. An AVDP at sector *N* or *N* - 256 shall not be recorded while a session is open. In an intermediate state, a single AVDP may exist at S + 256 or *S* + 512. An AVDP at *S* + 512 must be ignored when an AVDP exists at *S* + 256, *N* - 256 or *N*.

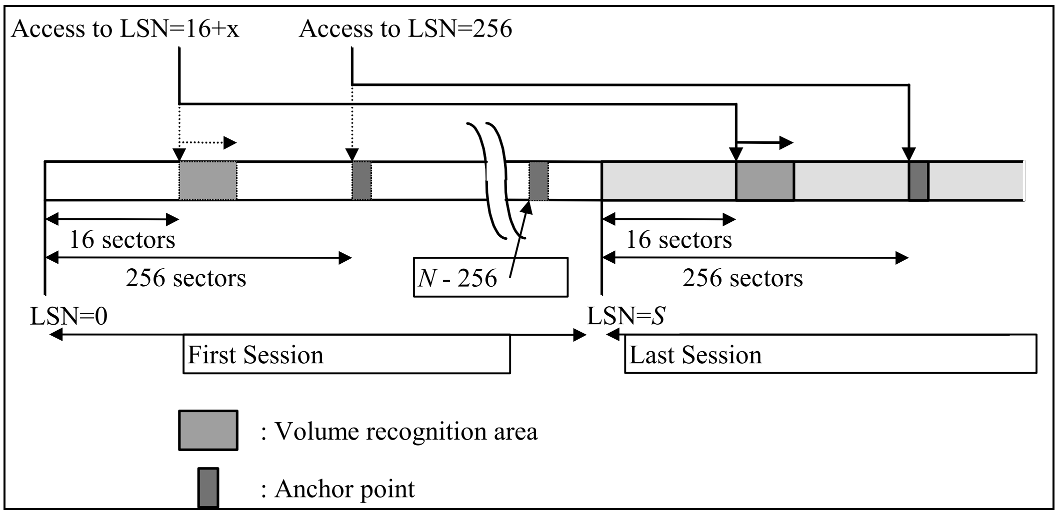
6.11.4 UDF Bridge format

The UDF Bridge format allows UDF to be added to a disc that may contain another file system. A UDF Bridge disc shall contain a UDF file system in its last session. The last session shall follow the rules described in 6.11.3. The disc may contain sessions that are based on ISO 9660, vendor unique, CD audio, or a combination of file systems.
ISO 9660 requires a Primary Volume Descriptor (ISO PVD) at sector 16 (assuming 2K sectors). If an ISO 9660 file system is desired, it may contain references to the same files as those referenced by ECMA 167 structures, or reference a different set of files, or a combination of the two.

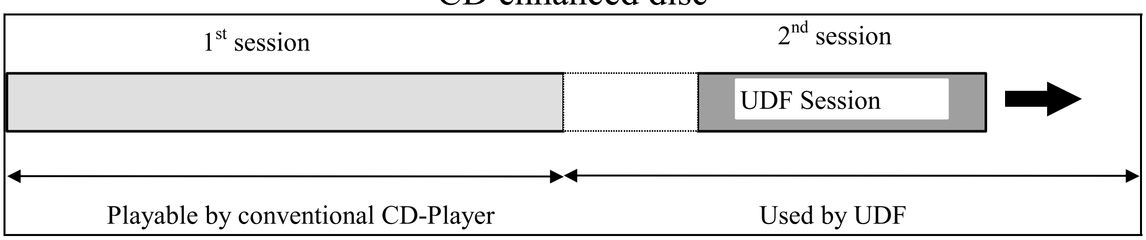
6.11.5 Examples of UDF Multisession and UDF Bridge

Some examples of UDF Multisession discs and UDF Bridge discs are shown below.

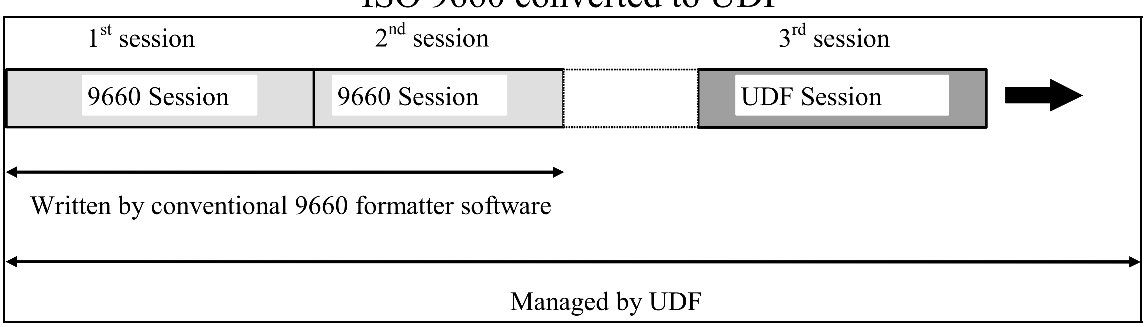
Multisession UDF disc



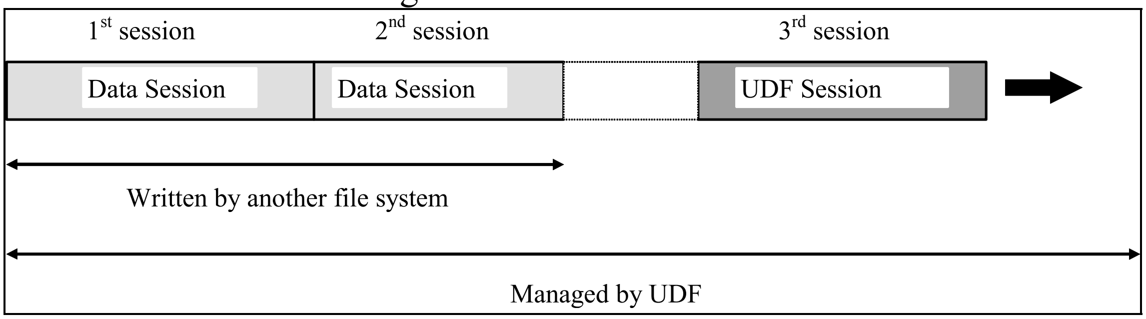
CD enhanced disc



ISO 9660 converted to UDF



Foreign format converted to UDF



6.12 Requirements for DVD-R/-RW/-RAM interchangeability

This appendix defines the requirements and restrictions on volume and file structures for writable DVD media, including but not limited to DVD-RAM discs (6.12.1), DVD-RW discs (6.12.2) and DVD-R discs (6.12.3), to support the interchange of information between users of both computer systems and consumer appliances. These requirements do not apply to the discs that are used in a computer system environment only and have no interchangeability with consumer appliances. The common requirements for these DVD discs are summarized as follows:

1. The volume and file structure shall comply with UDF 2.00.
2. The Minimum UDF Read Revision and Minimum UDF Write Revision shall be 2.00.
3. The length of logical sector and logical block shall be 2048 bytes.
4. A Main Volume Descriptor Sequence and a Reserve Volume Descriptor Sequence shall be recorded.

6.12.1 Requirements for DVD-RAM

The requirements for DVD-RAM discs are based on UDF 2.00. The volume and file structure is simplified as for Overwritable discs using non-sequential recording.

For Volume Structure:

1. A partition on a DVD-RAM disc shall be an overwritable partition specified as Access Type 4.
2. Virtual Partition Map and Virtual Allocation Table shall not be recorded.
3. Sparable Partition Map and Sparing Table shall not be recorded.

For File Structure:

4. Unallocated Space Table or Unallocated Space Bitmap shall be used to indicate a space set. Freed Space Table and Freed Space Bitmap shall not be recorded.
5. Non-Allocatable Space Stream shall not be recorded.

6.12.2 Requirements for DVD-RW

The requirements for DVD-RW discs under Restricted Overwrite mode are based on UDF 2.00. The volume and file structure is simplified as for Rewritable discs using non-sequential recording.

For Volume Structure:

1. A disc shall consist of a single volume with a single sparable partition per side.
2. A Sparable Partition Map and Sparing Table shall be recorded.
3. Length of a packet shall be 16 sectors (32 KB) and the first sector number of a packet shall be an integral multiple of 16.
4. Virtual Partition Map and Virtual Allocation Table shall not be recorded.

For File Structure:

5. Unallocated Space Bitmap shall be used to indicate a space set. Unallocated

Space Table, Freed Space Table and Freed Space Bitmap shall not be recorded.

6. Non-Allocatable Space Stream shall be recorded.
7. ICB Strategy Type 4 shall be used.
8. Short Allocation Descriptors or the embedded data shall be recorded in the Allocation Descriptors field of the File Entry or Extended File Entry. Long Allocation Descriptors shall not be recorded in this field.

6.12.3 Requirements for DVD-R

The requirements for DVD-R discs under Disc at once recording mode and under Incremental recording mode are based on UDF 2.00. The volume and file structure is simplified as for Write-Once discs using sequential recording.

For Volume Structure:

1. Length of a packet shall be an integral multiple of 16 sectors (32 KB) and the first sector number of a packet shall be an integral multiple of 16.
2. Sparable Partition Map and Sparing Table shall not be recorded.
3. Under Incremental recording mode, only one Open Integrity Descriptor shall be recorded in the Logical Volume Integrity Sequence.
4. Under Incremental recording mode, Virtual Partition Map shall be recorded.

For File Structure:

5. Unallocated Space Table, Unallocated Space Bitmap, Freed Space Table and Freed Space Bitmap shall not be recorded.
6. Only one File Set Descriptor shall be recorded.
7. Non-Allocatable Space Stream shall not be recorded.
8. Under Incremental recording mode, Virtual Allocation Table and VAT ICB shall be recorded.
9. Under Incremental recording mode, ICB Strategy Type 4 shall be used.
10. Under Incremental recording mode, the VAT entries in VAT shall be assigned as follows:
    - The virtual address 0 shall be used for File Set Descriptor.
    - The virtual address 1 shall be used for the ICB of the root directory.
    - The virtual addresses in the range of 2 to 255 shall be assigned for the File Entry of DVD_RTAV directory and File Entries of files under the DVD_RTAV directory.

6.12.4 Requirements for Real-Time file recording on DVD discs

DVD Video Recording specification defines the DVD specific sub-directory "DVD_RTAV" and all DVD specific files under the DVD_RTAV directory. DVD specific files consist of Real-Time files with the file type 249 and the related information files.

For Volume Structure:

1. For DVD-RAM/RW discs, a disc shall consist of a single volume with a single partition per side. For DVD-R discs, a disc shall consist of a single volume with a write-once partition and a virtual partition per side.
2. For DVD-RW discs, First Sparing Table and Second Sparing Table shall be recorded.

For File Structure:

3. For DVD-RAM/RW discs, only Unallocated Space Bitmap shall be used.
4. For DVD-RW discs, the extent of Unallocated Space Bitmap should have the length of Space Bitmap Descriptor for the available Data Recordable area.
5. Consumer Content Recorders record all their data in a special subdirectory, DVD_RTAV, located in the root directory. The DVD_RTAV directory and its contents have special file system restrictions which are defined in DVD Specifications published from DVD Format/Logo Licensing Corporation, see 6.9.3. An implementation or application should not create or modify files in this directory unless it meets the restrictions defined by DVD Specifications specified above.

6.13 Recommendations for DVD+R and DVD+RW Media

DVD+R and DVD+RW Media require special consideration due to their nature. The following information and guidelines are established to ensure interchange.

- Logical Sector Size is 2048 Bytes
- 2048 Bytes user data transfer for read and write
- ECC Block Size is 32768 bytes (32KB) and the first sector number of an ECC block shall be an integral multiple of 16.

6.13.1 Use of UDF on DVD+R media

For DVD+R, the rules of section 6.11 apply.

6.13.2 Use of UDF on DVD+RW 4.7 GBytes Basic Format media

DVD+RW 4.7 GBytes Basic Format media are random readable and writable, where needed the DVD+RW drive performs Read-Modify-Write cycles to accomplish this. For DVD+RW 4.7 GBytes Basic Format media the drive does not perform defect management. The DVD+RW 4.7 GBytes Basic Format provides the following features:

- Random read and write access
- Background physical formatting
- The Media Type is Overwritable (partition Access Type 4, overwritable)

6.13.2.1 Requirements

- Sparing shall be managed by the host via the Sparable Partition and a Sparing Table.
- The sparing Packet Length shall be 16 sectors (32 KB, one ECC block).
- Defective packets known at format time shall be allocated by the Non-Allocatable Space Stream, see 3.3.7.2.

6.13.2.2 Background Physical Formatting

Physical formatting is performed by the drive in background. In implementing the host applications, the following requirements for the drive should be considered:

- After some minimal amount of formatting has been performed, the operation continues in background.
- At the initialization of the file system, after the Background Physical Formatting has been started, the host must record the first AVDP at sector 256. The second AVDP must be recorded after the Background physical Formatting has been finished. Before the second AVDP has been recorded, the file system is in an intermediate state and is not strictly in compliance with ECMA 167. The disc can be ejected before the background formatting has finished, and in that case only one AVDP exists. Note that at an early eject the drive must format all non-recorded areas up to the highest sector number recorded by the host, this could cause a significant delay in the early eject process. Implementations are recommended to allocate the lowest numbered blocks available while background physical formatting is in progress.
- The background physical formatting status shall not influence the recording of the LVID. At early eject the LVID shall be recorded in the same way as it will be recorded on Rewritable media that do not support background physical formatting.

The physical formating may be followed by a verification pass. Defects found during the verification pass shall be enumerated in the *Non-Allocatable Space Stream*, see 3.3.7.2.

Finally, file system root structures shall be recorded. These mandatory file system and root structures include the Volume Recognition Sequence, the Anchor Volume Descriptor Pointers, the Volume Descriptor Sequences, a File Set Descriptor and a Root Directory.
Allocation for sparing shall occur during the formatting process. The sparing allocation may be zero in length.

The unallocated space descriptors shall be recorded and shall reflect the space allocated to not-spared defective areas and sector sparing areas.

The format may include all available space on the medium. However, formatting may be interrupted upon request by the user. Formatting may later be continued to the full space.

6.14 Recommendations for Mount Rainier formatted media

The following guidelines are established to ensure interchange of Mount Rainier (MRW) formatted media.

6.14.1 Properties of CD-MRW and DVD+MRW media and drives

The following is a list of key properties of MRW media and drives:

- A Physical Sector Size of 2048 Bytes
- The drive performs Read/Modify/Write cycles when needed. Data transfer between the host and the MRW drive is in multiples of 2048 bytes.
- Random access read and write is possible
- Drive level defect management
- The drive performs background physical formatting
- The Media Type is Overwritable (partition Access Type 4, overwritable)
- A Non-Allocatable Space List, Non-Allocatable Space Stream and Sparing Table shall not be used on MRW formatted media

6.14.2 Background Physical Formatting

At the initialization of the file system, after the Background Physical Formatting has been started, the host must record the first AVDP at sector 256. The second AVDP must be recorded after the Background physical Formatting has been finished. Before the second AVDP has been recorded, the file system is in an intermediate state and is not strictly in compliance with ECMA 167. The disc can be ejected before the background formatting is finished, in that case only one AVDP exists on the MRW disc. Note that at an early eject the drive must format all non-recorded areas up to the highest sector number recorded by the host, this could cause a significant delay in the early eject process. Implementations are recommended to allocate the lowest numbered blocks available while background physical formatting is in progress.

The background physical formatting shall not influence the recording of the LVID. At early eject the LVID shall be recorded in the same way as it will be recorded on Rewritable media that do not support background physical formatting.

6.15 Introduction to the Pseudo OverWrite Mechanism

In previous UDF revisions (as described in UDF 1.50 through 2.50), multiple sessions, or the VAT is used to achieve sequential recording functionality on CD-R, DVD-R, and DVD+R media. Next generation drives supporting *pseudo overwrite capability* on sequentially recordable media will contribute to a decrease in file system complexity. The UDF Pseudo OverWrite method described in this appendix can be applied to such pseudo-overwritable sequentially recordable media.

Benefits of the UDF Pseudo OverWrite method include:

- Increased compatibility as ensured by the drive supporting pseudo overwrite functionality and defect management
- Reduced complexity in file system implementations since the entire volume space is Overwritable (at logical sector granularity) while defect management is implemented in the drive
- UDF implementations can use the Metadata File to locate metadata in a logically contiguous manner. This metadata can optionally be duplicated in the Metadata Mirror File in order to achieve the desired redundancy

6.15.1 Characteristics of Media formatted for Pseudo OverWrite

Media formatted for Pseudo OverWrite will support multi-track recording. All logical sectors in the volume space on the media can be overwritten.

The file system can write concurrently to multiple tracks. A track is defined as *reserved* or *used*, see 1.3.2. Each track is sequentially recordable only. The *Next Writable Address* (or NWA) is obtained by the file system by querying the drive and points to the next recordable logical sector within the track.

In addition to sequential recording, any logical sector in a track before the NWA can be independently overwritten. Also sectors in a *used* track (having no valid NWA) can be overwritten. Overwriting is supported by the drive by recording updated data either within the Spare Area (by the linear replacement algorithm) or to some NWA within the volume space. UDF does not currently propose any policy specifiable by the file system to control physical placement of data being overwritten. While performing sequential recording on the medium after requesting the NWA of a track, the drive system shall behave in such a way that the NWA will not change unexpectedly, or without notification, until the UDF implementation queries for the NWA of that track again. When pseudo overwrite is performed all the NWAs become invalid.

The drive is entirely responsible for maintaining the remap entry information for the logical sectors that can and may be persisted within the volume space.

6.15.2 Write Strategy

Tracks can be utilized to record different data types in a logically contiguous manner (e.g. metadata, metadata mirror and data, can be recorded in separate tracks). When all unrecorded sectors in a *reserved* track have been exhausted, the UDF implementation can assign a new *reserved* track (by *splitting* any existing *reserved* track) of an appropriate size.

By allowing *reserved* tracks to be split, the drive enables recording of the AVDP (comprising volume structure) at any two locations of: LSN 256, the last LSN in the volume, or (Last LSN – 256) as per ECMA 167.

It is desirable for UDF implementations to duplicate the metadata in the Metadata Mirror File.

Figure 1 below illustrates the track layout for a freshly formatted medium where the Metadata Mirror File is not being recorded. Track #1 contains the volume structure (including the AVDP at LSN 256) as well as related file structures. The Duplicate Metadata Flag in the Metadata Partition Map is set to zero. The format utility has allocated an extent (track) for metadata recording while Track #3 comprises the majority of recordable volume space to be utilized as required.

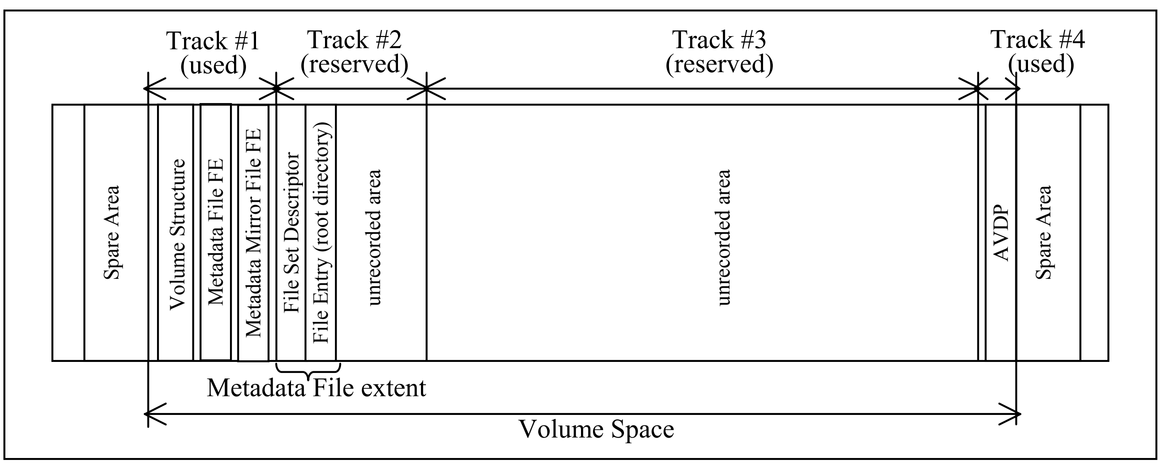


Figure 1: Freshly formatted medium – no Metadata Mirror File

Figure 2 below also illustrates the track layout for a freshly formatted medium where the Metadata Mirror File will be recorded. The Duplicate Metadata Flag in the Metadata Partition Map is set to one. Hence an extent has been allocated for the contents of the Metadata Mirror File.

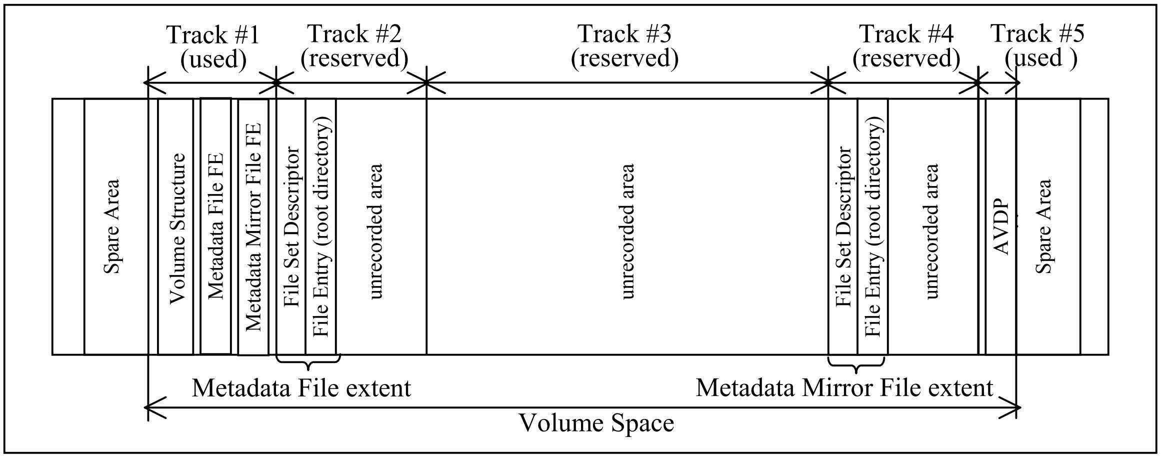


Figure 2: Freshly formatted medium – Metadata Mirror File will be recorded

Figure 3 below illustrates track layout on media after files have been recorded (note that – in this case – the Metadata Mirror File is not being recorded). In this illustration, Track #2 is in the used state; hence Track #4 was allocated for additional recording of metadata (Track #3 is being used to record data).

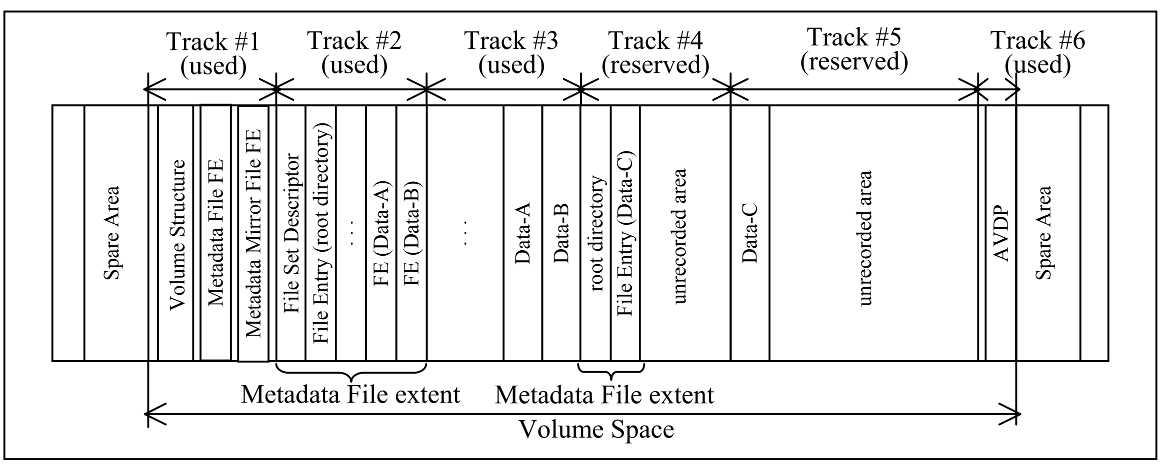


Figure 3: Recording data on medium (no Metadata Mirror File)

6.15.3 Requirements for UDF Implementations

UDF implementations are expected to conform to the following requirements:

- For sequentially recordable media formatted for Pseudo OverWrite, the Access Type in the Partition Descriptor shall be set to zero (pseudo-overwritable), see section 2.2.14.2
- The Unallocated Space Bitmap and Unallocated Space Table shall not be recorded
- The Metadata Partition Map shall be recorded
- The Metadata Bitmap File shall not be recorded
- Up to 4 tracks can be concurrently in a "reserved" state
- Multisession/Multiborder recording shall not be used with Pseudo OverWrite

6.15.4 Implementation Notes for UDF Implementations

- Query the drive to determine whether a pre-formatted medium supports Pseudo OverWrite. At format time, set the pseudo overwrite attribute on the medium (as per UDF implementation policy).
- Writing data to previously unrecorded sectors will require querying the drive to determine the NWA in a track – the returned value will be an absolute logical sector number (relative to LSN 0 in the volume space).
- Do not attempt to re-use sectors previously allocated to a file marked for deletion.
- Minimize the amount of data being overwritten.
- Prior to allocating a new *reserved* track (by splitting an existing *reserved* track) ensure that the current track reserved for such data/metadata is in the *used* state.
- The Metadata File and the Metadata Mirror File can have more than one extent in a single track. The extents of these files should not be pre-allocated as some of the sectors could be used by the drive for Pseudo OverWrite or defect management.

6.16 Recommendations for Blu-ray Disc media

This appendix defines the requirements and recommendation on volume and file structures for Blu-ray Disc (BD) media, to support data interchange among computer systems and consumer appliances. These requirements do not apply to the discs when the use of the discs is limited to computer systems and there is no necessity to provide interchangeability with consumer appliances. Specific requirements related to BDAV and BDMV application usage are described in section 6.16.4.

Blu-ray Disc has the following three types of media:

- Blu-ray Disc Read-Only Format (BD-ROM)
- Blu-ray Disc Rewritable Format (BD-RE)
- Blu-ray Disc Recordable Format (BD-R)

BD-R can use either SRM with LOW or SRM without LOW, for details see section 6.16.3. BD-ROM, BD-RE and BD-R using SRM without LOW, all use UDF revision 2.50. BD-R using SRM with LOW uses UDF revision 2.60, rather than 2.50.
Common characteristics and requirements for these three media types are:

1. Logical sector size is 2048 bytes.
2. ECC Block Size is 65536 bytes (64KB)
3. Sparable Partition Map and Sparing Table shall not be recorded.
4. Non-Allocatable Space Stream shall not be recorded.

6.16.1 Requirements for Blu-ray Disc Read-Only Format (BD-ROM)

A Blu-ray Read-Only disc (BD-ROM) is a Read-Only medium.
The BD-ROM File System Format shall comply with UDF revision 2.50 and has the following additional requirements:

For Volume Structure:

1. The Partition Descriptor Access Type shall be 1 (read-only).
2. Three Anchor Volume Descriptor Pointers should be recorded.

For File Structure:

3. Unallocated Space Table and Unallocated Space Bitmap shall not be recorded.
4. Metadata Bitmap File shall not be recorded.

NOTE: Duplication of Metadata File data is optional. When robustness is required, it is recommended that duplication is used and that Metadata File and Metadata Mirror File data and descriptors are recorded at the physically inner radius area and outer radius area, respectively.

6.16.2 Requirements for Blu-ray Disc Rewritable Format (BD-RE)

A Blu-ray Rewritable disc (BD-RE) is a non-sequential recording medium. A BD-RE drive performs read modify write operations when needed. Defect free logical space is provided by a BD-RE drive which performs defect management using the linear replacement algorithm.
The BD-RE File System Format shall comply with UDF revision 2.50 and has the following additional requirements:

For Volume Structure:

1. The Partition Descriptor Access Type shall be 4 (overwritable).

For File Structure:

2. An Unallocated Space Bitmap shall be recorded, no Unallocated Space Table.

NOTE 1: Duplication of Metadata File data is optional. When the user requires robustness rather than write performance, it is recommended that duplication is used and that Metadata File and Metadata Mirror File data and descriptors are recorded at the physically inner radius area and outer radius area, respectively.

Requirements for Defect Management:

Spare Area shall be assigned on a Blu-ray Rewritable disc, as the UDF file system requires Drive Defect Management by the drive system. In general, Spare Areas with the default size are assigned at format time.

NOTE 2: When the available clusters in Spare Area are exhausted, additional Spare Area can be allocated after all data is backed up to the other media. On the other hand, if a special utility tool can move some file data and volume structure on the disc in order to shorten the volume space, the Spare Area can be expanded preserving the file data on the disc.

6.16.3 Requirements for Blu-ray Disc Recordable Format (BD-R)

A Blu-ray Recordable disc (BD-R) is a Write-Once medium that can use Sequential Recording Mode (SRM) either with or without Logical OverWrite (LOW). Drive based defect management using the linear replacement algorithm is supported.
The Pseudo OverWrite (POW) Method as described in 6.15 can be applied on BD-R media formatted using SRM with LOW.

The BD-R File System Format shall comply with UDF revision 2.60 for SRM with LOW (POW) and shall comply with UDF 2.50 for SRM without LOW (non-POW). The following additional requirements are applied:

For Volume Structure:

1. For SRM with LOW, the Partition Descriptor Access Type shall be 0 (pseudo-overwritable).
2. For SRM without LOW, the Partition Descriptor Access Type shall be 1 (read-only) or 2 (write-once).

For File Structure:

3. Unallocated Space Table and Unallocated Space Bitmap shall not be recorded.
4. Only ICB Strategy Type 4 shall be used.

Requirements for Defect Management:

Spare Area shall be assigned for a BD-R medium formatted for SRM with LOW (POW). In general, Spare Areas with the default size are assigned at format time.

6.16.4 Information about AV Applications

The Blu-ray Disc Format has two types of AV Application Formats that are called "BDAV Application" and "BDMV Application".

Information about BDAV Application Use

The "BDAV Application" is a Video Recording Format for BD-RE discs and BD-R discs, including AV Stream and database for playback the AV Stream.
The "BDAV", "BDAV1", "BDAV2", "BDAV3", and "BDAV4" directories immediately under the root directory are reserved for the BDAV application.

Information about BDMV Application Use

The "BDMV Application" is a Video Application Format for BD-ROM discs, including AV Stream and database for playback the AV Stream.
The "BDMV" directory immediately under the root directory is reserved for the BDMV application.

6.16.4.1 Requirements for BDAV and BDMV Application usage

The following additional requirements are applied for BDAV and BDMV Application usage:

1. A volume set shall consist of only one volume.
2. Only one prevailing Partition Descriptor shall be recorded in the Volume Descriptor Sequence.
3. A Metadata Partition Map shall be recorded.
4. Symbolic Links shall not be used for all files and directories (the value of the File Type field in the ICB shall not be 12).
5. Hard Link shall not be used for all files and directories.
6. Multisession and VAT recording shall not be used.

6.17 UDF Media Format Revision History

The following table shows when changes to the UDF Specification have taken place that affect the UDF format that can be recorded on a piece of media. The Document Change Notices (DCNs), which document a specific change, are referenced in the table. The column *Update in UDF Revision* describes which revision of the UDF specification that the change was included. The fields *Minimum UDF Read Revision* and *Minimum UDF Write Revision* relate to the Revision Access Control fields described in 2.2.6.4.

Description	DCN	Updated in UDF Revision	Minimum UDF Read Revision	Minimum UDF Write Revision
UDF 1.02				
Allocation Extent Descriptor	2-002	1.02	1.02	1.02
Path Component File Version Number	2-003	1.02	1.02	1.02
Parent Directory Entries	2-004	1.02	1.02	1.02
Device Specification Extended Attribute	2-005	1.02	1.01	1.02
Maximum Logical Extent Length	2-006	1.02	1.02	1.02
Unallocated Space Entry	2-008	1.02	1.01	1.02
DVD Copyright Management Information	2-009	1.02	1.02	1.02
Logical Volume Identifier	2-010	1.02	1.01	1.02
Extent Length Field of an Allocation Descriptor	2-012	1.02	1.01	1.02
Non-relocatable & Contiguous Flags	2-013	1.02	1.01	1.02
Revision of Requirements for DVD-ROM	2-014	1.02	1.02	1.02
Revision Access Control	2-015	1.02	1.01	1.02
Volume Set Identifier	2-017	1.02	1.01	1.02
UniqueIDs for Extended Attributes	2-018	1.02	1.02	1.02
Clarification of Dstrings	2-019	1.02	1.01	1.02
Application FreeEASpace Extended Attribute	2-020	1.02	1.02	1.02
Update of Identifier Suffix to 1.02	2-021	1.02	1.02	1.02
UDF 1.50				
Update of Identifier Suffix to 1.50	2-025	1.50	1.50	1.50
Virtual Partition Map Entry	2-026	1.50	1.50	1.50
Allocation of Sparable Partition Map	2-027	1.50	1.50	1.50
Addition of Virtual Allocation Table	2-028	1.50	1.50	1.50
Addition of Sparing Table	2-029	1.50	1.50	1.50
Addition of Non-Allocatable Space List	2-030	1.50	1.02	1.50
Reccommmendations for CD Media	2-031	1.50	1.50	1.50

Desciption	DCN	Updated in UDF Revision	Minimum UDF Read Revision	Minimum UDF Write Revision
UDF 2.00				
Change 1.50 to 2.00	2-033	2.00	1.02	2.00
Clarified Domain flags	2-034	2.00	1.02	2.00
Unicode 2.0 Support	2-035	2.00	1.02	2.00
Named Streams	2-036	2.00	2.00	2.00
Unique ID Table as a Named Stream	2-037	2.00	1.02	2.00
Mac Resource Fork as a Named Stream	2-038	2.00	2.00	2.00
Location Field of the Extended Attribute Header	2-043	2.00	1.02	2.00
Access Control Lists	2-044	2.00	2.00	2.00
Descriptor Tags spanning block boundaries	2-047	2.00	1.02	2.00
Power Calibration Stream	2-048	2.00	1.02	2.00
Support for CD-R Multisession Required	2-050	2.00	1.50	2.00
Value of fields in LVID for virtual partition on CD-R	2-051	2.00	1.50	2.00
System Stream to indicate volume backup time	2-055	2.00	2.00	2.00
New VAT	2-056	2.00	2.00	2.00
Restricting Virtual Addresses	2-057	2.00	1.50	2.00
File Times Extended Attribute	2-058	2.00	1.02	2.00
OS/2 EA Stream	2-061	2.00	2.00	2.00
Non-Allocatable Space Stream	2-062	2.00	1.02	2.00
UDF 2.01				
Tag serial number & disaster recovery	5000	2.01	1.02	1.02
Change to DOS name transform algorithm	5002	2.01	-	1.02
Directory search order for dual namespaces	5004	2.01	1.02	1.02
Termination in strategy 4096 clarification	5006	2.01	1.02	1.02
Compression Ids 254/255 clarification	5007	2.01	2.00	2.00
Mac Resource Fork can only be in files	5008	2.01	2.00	2.00
Requirements for CD media	5009	2.01	1.50	1.50
AVDP Placement	5013	2.01	1.50	1.50
Non relocatable bit clarification	5014	2.01	1.02	1.02
Various editorial corrections	5015	2.01	-	-
PCA stream fix	5018	2.01	2.00	2.00
Parent of system stream directory	5019	2.01	2.00	2.00
OS/400 updates	5020	2.01	2.00	2.00
Missing EntityID definitions	5021	2.01	2.00	2.00
Various editorial corrections	5024	2.01	-	-
New OS types	5025	2.01	2.00	2.00
PVD Application Identifier field clarification	5026	2.01	1.02	1.02
Descriptor CRC length	5027	2.01	1.02	1.02
POSIX permissions clarifications	5029	2.01	2.00	2.00
Clarification of 3,2,1,1	5030	2.01	2.00	2.00
Volume recognition sequence	5031	2.01	1.02	2.00
Path length	5032	2.01	1.02	1.02
FID LengthOfImplementationUse	5034	2.01	1.02	1.02
Editorial – non-allocatable space stream	5035	2.01	-	-
Allocation extent descriptor CRC length	5036	2.01	2.00	2.00
File types 248 to 255	5037	2.01	2.00	2.00
Real-time files on DVD-RAM	5038	2.01	2.00	2.00
Packet length specification	5039	2.01	2.00	2.00
Overlapping structures with conflicting field	5040	2.01	2.00	2.00
Information length reconstruction	5041	2.01	2.00	2.00
Timezone interpretation	5042	2.01	1.02	1.02
Missing partition descriptor and sparable partition	5044	2.01	1.02	1.02
Basic restrictions & requirements PD correction	5045	2.01	1.50	1.50
PVD and LVD volume sequence number	5046	2.01	1.02	1.02
Additions to 5.1 informative table	5047	2.01	2.00	2.00
Clarify uniqueID use for EAs/streams	5048	2.01	2.00	2.00

Description	DCN	Updated in UDF Revision	Minimum UDF Read Revision	Minimum UDF Write Revision
UDF 2.50				
FID File Identifier length and Unicode uniqueness	5049	2.50	1.02	2.01
Disallow overlapping partitions	5061	2.50	1.02	1.02
Strategy 4096 only for WORM media	5062	2.50	1.02	1.02
UDF Unique ID Mapping Data	5063	2.50	2.50	2.50
Extended Attribute block alignment	5064	2.50	1.02	1.02
UDF Defined Named Streams section	5065	2.50	2.00	2.00
File Identifier translation code repair	5066	2.50	1.02	1.02
Correction of is_fileset_soft_protected rule	5069	2.50	2.00	2.00
Disallow hard linked directories	5070	2.50	1.02	2.50
Requirements for DVD-RAM/RW/R interchangeability	5071	2.50	2.00	2.00
Unique ID for System Stream Directory	5072	2.50	2.50	2.50
Shared description for some LVID and VAT fields	5074	2.50	2.01	2.01
Recommendations for Mount Rainier formatted media	5075	2.50	1.02	1.02
Recommendations for DVD+R and DVD+RW	5076	2.50	1.50	1.50
Section 3.3.6 put out of order	5077	2.50	2.00	2.00
UDF UniqueID clarifications	5078	2.50	2.00	2.00
Clarify partition Access Type 3 and 4	5079	2.50	2.01	2.01
Icbtag Parent ICB Location issue	5081	2.50	1.02	2.50
Clarification of Volume Recognition Sequence	5082	2.50	1.02	2.01
Metadata Partition Map	5086	2.50	2.50	2.50
Partition Alignment & ECC Block Size Definition	5089	2.50	1.02	2.50
Non-allocatable space stream usage clarifications	5090	2.50	1.50	1.50
UDF 2.60				
Editorial corrections for UDF revision after UDF 2.50.	5100	2.60	-	-
Virtual, metadata and read-only partitions	5101	2.60	2.50	2.50
No Metadata Bitmap File required for read-only	5102	2.60	2.50	2.50
Equivalence for Metadata File and Mirror File	5103	2.60	2.50	2.50
Next extent for Metadata File and Metadata Mirror File	5104	2.60	2.50	2.50
Terminating Descriptor in Metadata Partition	5105	2.60	2.50	2.50
Metadata Mirror File FEs and AEDs always far apart	5106	2.60	2.50	2.50
Clarify overlapping of Sparing Table with a partition	5107	2.60	1.50	1.50
Descriptor CRC Length Uint16 overflow rules	5108	2.60	1.02	1.02
Clarification of NOTE on page 41	5109	2.60	2.50	2.50
Appoint OS Identifier for UNIX - NetBSD	5110	2.60	1.02	1.02
Pseudo OverWrite Method	5111	2.60	2.60	2.60
BD non-POW media recommendations for UDF 2.50	5112	2.60	2.50	2.50
Main and Reserve VDS far apart	5113	2.60	2.50	2.50
BD-R recommendations for UDF 2.60	5114	2.60	2.60	2.60
Enable UDF 2.50 POW read compatibility	5115	2.60	2.50	2.50
Consequences of Pseudo OverWrite Method	5116	2.60	2.60	2.60
Common aspects of recording for different media	5117	2.60	2.01	2.01
Clarify location of Partition Header Descriptor	5118	2.60	1.02	1.02
Zero Inf. Length for Non-Allocatable Space Stream	5119	2.60	2.60	2.60
Minimum UDF Read Revision for UDF 2.60 media	5120	2.60	2.60	2.60
Clarification of Directory bit in parent FID	5121	2.60	2.00	2.00

6.18 Developer Registration Form

Any developer that plans on implementing ECMA 167 according to this document should complete the Developer Registration Form on the following page. By becoming a registered OSTA developer you receive the following benefits:

- You will receive a list of the current OSTA registered developers and their associated *Developer IDs*. The developers on this list are encouraged to interchange media to verify data interchange among implementations.
- Notification of OSTA UDF Committee meetings. You may attend a limited number of these meetings without becoming an official OSTA member.
- You can be added to the OSTA UDF email reflector. This reflector provides you the opportunity to post technical questions on the *OSTA Universal Disk Format Specification*.
- You will receive an invitation to participate in the development of the next revision of this document.

Note 2 in section 2.1.5.2 explains how a Developer ID should look like.

For the latest information on OSTA and UDF visit the OSTA web site, see **POINTS OF CONTACT** on the first page of this document.

The Developer Registration Form is printed on the next page of this document.

OSTA Universal Disk Format Specification Developer Registration Form

Optical Storage Technology Association

Name: ______________________________

Company: ______________________________

Address: ______________________________

____

____

City: ______________________________

State/Province: ______________________________

Zip/Postal Code: ______________________________

Country: ______________________________

Phone: ______________ FAX: ______________

Email: ______________________________

Please indicate on which operating systems you plan to support UDF:

O DOS	O OS/2	O Macintosh	O Linux	
O UNIX/POSIX	O OS/400	O Windows 9x	O Windows NT/2000	O Windows XP

O Other ______________________________

Please indicate which media types you plan to support:

O Magneto Optical	O WORM	O Phase Change	
O CD-ROM	O CD-R	O CD-RW	O CD-MRW
O DVD-ROM	O DVD-R	O DVD-RW	O DVD-RAM
O DVD-Video	O DVD-Audio		
O DVD+RW	O DVD+R	O DVD+MRW	
O BD-ROM	O BD-RE	O BD-R	
O HD DVD-ROM	O HD DVD-Rewritable	O HD DVD-R	

O Other ______________________________

Please indicate what value you plan to use as EntityID "*Developer ID" to identify your implementation. The Developer ID should uniquely identify your company as well as your product, see section 2.1.5.2 note 2 in the latest UDF specification.

____

O Please add my email address to the OSTA UDF email reflector.
O Please send an OSTA Membership kit.

Send completed form to OSTA. For address, see http://www.osta.org/osta/contact.htm.

P

R

S

T

U

V

W