-OSTA-

OSTA Universal Disk Format Specification
Revision 1.50

===========================Readme============================

The pages included in this distribution are as follows: Page#

UDF_150.PDF	Adobe Acrobat format of UDF Specification	2-119
HISTORY	Revision history of UDF Specification	120
UNICODE.C	Unicode sample source code	121-122
DOSNAME.C	UDF DOS filename translation	123-127
UDFTRANS.C	UDF OS/2, Macintosh and UNIX filename translation	128-131
FILE_ID.DIZ	BBS Description file	132

********** I M P O R T A N T   N O T E **********
Please fill out the OSTA UDF Developer Registration form located in the UDF specification and return it to OSTA. This will make sure that you are kept up to date with announcements in regards to UDF.

For additional information on OSTA and UDF visit the OSTA web site at http://www.osta.org.

NOTE: Free Adobe Acrobat readers for several platforms are available at http://www.adobe.com

Universal Disk Format™ Specification

Revision 1.50

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 (DCN 2-025 through DCN 2-032)

Important Notices

CONTENTS

iii

This page left intentionally blank

1. Introduction

The OSTA Universal Disk Format (UDF™) specification defines a subset of the standard ISO/IEC 13346. The primary goal of the OSTA UDF is to maximize data interchange and minimize the cost and complexity of implementing ISO/IEC 13346.

To accomplish this task this document defines a *Domain*. A domain defines rules and restrictions on the use of ISO/IEC 13346. 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 ISO/IEC 13346 on a per operating system basis:

> *Given some ISO/IEC 13346 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 ISO/IEC 13346 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 ISO/IEC 13346 standard easier.

To be informed of changes to this document please fill out and return the OSTA UDF Developers Registration Form located in appendix 6.11.

1.1 Document Layout

This document presents information on the treatment of structures defined under standard ISO/IEC 13346.

This document is separated into the following 4 basic sections:

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

This document presents information on the treatment of structures defined under standard ISO/IEC 13346. 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 ISO/IEC 13346, 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:"**

1.2   Compliance

This document requires conformance to parts 1, 2, 3 and 4 of ISO/IEC 13346. Compliance to part 5 of ISO/IEC 13346 is not supported by this document. Part 5 may be supported in a later revision of this document.

NOTE: Due to the nature of CD media, Partitions may contain volume structures. This violates ISO 13346 (3/8.5). Efforts are under way to revise ISO 13346 to allow volume structures within write-once partitions.

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 accessable.
- *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.

The full definition of compliance to this document is defined in a separate OSTA document.

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. References enclosed in [ ] in this document are references to ISO 13346. 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.
ECMA 167	European Computer Manufactures Association (ECMA) standard number 167. Revision 2 of this standard is equivalent to ISO/IEC 13346:1995, and is available from http://www.ecma.ch.

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 the 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 the ISO/IEC 10149.
Dirty File System	A file system that is not a clean file system.
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 ISO 13346.
Logical Block Address	An address relative to the beginning of a partition, as defined in ISO 13346.
Media Block Address	The address of a sector as it appears on the medium, before any mapping performed by the device.
Packet	A recordable unit, which is an integer number of sectors.
Packet Size	The number of user data sectors in a Packet.
Physical Address	An address used when accessing the medium, as it would appear at the interface to the device.
Random Access File System	A file system for randomly writable media, either write once or rewritable
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 as specified by 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
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.
VAT ICB	A File Entry ICB that describes a file containing a Virtual Allocation Table.
Virtual Address	An address described by a Virtual Allocation Table entry.
VAT	The Virtual Allocation Table (VAT) provides a Logical Block Address for each Virtual Address. The Virtual Allocation Table is used with sequential write once media.

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.

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 ISO 13346 structures. This area shall not be referenced by the Unallocated Space Descriptor or any other ISO 13346 descriptor. This is intended for use by the native operating system.
Volume Recognition Sequence	The Volume Recognition Sequence as described in part 2 of ISO/IEC 13346 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, except for the Space Bitmap Descriptor.
File Name Length	Maximum of 255 bytes
Maximum Pathsize	Maximum of 1023 bytes
Extent Length	Maximum Extent Length shall be 2^{30} - *Logical Block Size*
Primary Volume Descriptor	There shall be exactly one prevailing Primary Volume Descriptor recorded per volume.
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.
Partition Descriptor	A Partition Access Type of Read-Only, Rewritable, Overwritable and WORM 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 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 or Overwritable. 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 a 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.
Logical Volume Integrity Descriptor	Shall be recorded.
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 on Rewritable/Overwritable media. For WORM media multiple File Set Descriptors may be recorded based upon certain restrictions defined in this document.
ICB Tag	Only 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 *Allocation Extent Descriptor* 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.
Space Bitmap Descriptor	CRC not required.
Partition Integrity Entry	Shall not be recorded.
Volume Descriptor Sequence Extent	Both the main and reserve volume descriptor sequence extents shall each have a minimum length of 16 logical sectors.
Record Structure	Record structure files, as defined in part 5 of ISO/IEC 13346, shall not be created.

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 1.1 standard (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*.
17-255	Reserved

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 1.1 standard. Refer to appendix on *OSTA Compressed Unicode* for sample C source code to convert between *OSTA Compressed Unicode* and standard Unicode 1.1.

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

2.1.2 OSTA CS0 Charspec

```
struct Charspec {
       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 ISO 13346 standard, as well as this document, has normally defined byte positions relative to 0. In section 7.2.12 of ISO 13346, 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 {       /* ISO 13346 1/7.3 */
        Uint16          TypeAndTimezone;
        Uint16          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.

☞ 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.

☞ 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 this field. Otherwise the time zone portion of this field shall be set to -2047.

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

2.1.5 Entity Identifier

```
struct EntityID {       /* ISO 13346 1/7.4 */
        Uint8           Flags;
        char            Identifier[23];
        char            IdentifierSuffix[8];
}
```

UDF classifies *Entity Identifiers* into 3 separate types as follows:

- *Domain Entity Identifiers*
- *UDF Entity Identifiers*
- *Implementation Entity Identifiers*

The following sections describes the format and use of *Entity Identifiers* based upon the different types mentioned above.

2.1.5.1 Uint8 Flags
☞   Self explanatory.

✍   Shall be set to ZERO.

2.1.5.2 char Identifier
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 ISO 13346 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
Implementation Use Volume Descriptor	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Implementation Use Volume Descriptor	Implementation ID	"*UDF LV Info"	UDF Identifier Suffix
Partition Descriptor	Implementation ID	*"*Developer ID"*	Implementation 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 ID	*"*Developer ID"*	Implementation Identifier Suffix *(optional)*

File Entry	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
UDF Extended Attribute	Implementation ID	*See Appendix*	UDF Identifier Suffix
Non-UDF Extended Attribute	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Device Specification Extended Attribute	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Logical Volume Integrity Descriptor	Implementation ID	*"*Developer ID"*	Implementation Identifier Suffix
Partition Integrity Entry	Implementation ID	N/A	N/A
Virtual Partition Map	Partition Type Identifier	"*UDF Virtual Partition"	UDF Identifier Suffix
Sparable Partition Map	Partition Type Identifier	"*UDF Sparable Partition"	UDF Identifier Suffix
Virtual Allocation Table	Entity ID	"*UDF Virtual Alloc Tbl"	UDF Identifier Suffix
Sparing Table	Sparing Identifier	"*UDF Sparing Table"	UDF Identifier Suffix

> *NOTE:* 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 the appendix.

In the *ID Value* column in the above table *"*Developer ID"* refers to a 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:** 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.

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 paragraphs.

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

2.1.5.3 IdentifierSuffix

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

In regard to OSTA Domain *Entity Identifiers* specified in this document (appendix 6.1) the *IdentifierSuffix* field shall be constructed as follows:

Domain *IdentifierSuffix* field format

RBP	Length	Name	Contents
0	2	UDF Revision	Uint16 (= #0150)
2	1	Domain Flags	Uint8
3	5	Reserved	bytes (= #00)

The *UDFRevision* field shall contain **#0150** to indicate revision **1.50** 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	Hard Write-Protect
1	Soft Write-Protect
2-7	Reserved

The *SoftWriteProtect* flag is a user settable flag that indicates that the volume or file system 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. These flags are only used in the Logical Volume Descriptor and the File Set Descriptor. The flags in the Logical Volume descriptor have precedence over the flags in the File Set Descriptors.

Implementation use *Entity Identifiers* defined by UDF (appendix 6.1) the *IdentifierSuffix* field shall be constructed as follows:

UDF *IdentifierSuffix*

RBP	Length	Name	Contents
0	2	UDF Revision	Uint16 (= #0150)
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 the Appendix on *Operating System Identifiers*.

For implementation use *Entity Identifiers* not defined by UDF the *IdentifierSuffix* field shall be constructed as follows:

Implementation *IdentifierSuffix*

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 which 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.

2.2 Part 3 - Volume Structure
2.2.1 Descriptor Tag

```
struct tag {              /* ISO 13346 3/7.2 */
        Uint16    TagIdentifier;
        Uint16    DescriptorVersion;
        Uint8     TagChecksum;
        byte      Reserved;
        Uint16    TagSerialNumber;
        Uint16    DescriptorCRC;
        Uint16    DescriptorCRCLength;
        Uint32    TagLocation;
}
```

2.2.1.1 Uint16 TagSerialNumber

 ☞ Ignored. Intended for disaster recovery.

 ✍ Reset to a unique value at volume initialization.

The *TagSerialNumber* shall be set to a value that differs from ones previously recorded, upon volume re-initialization.   It is suggested that: *TagSerialNumber* = ((*TagSerialNumber* of the Primary Volume Descriptor) + 1).

2.2.1.2 Uint16 DescriptorCRCLength

CRCs shall be supported and calculated for each descriptor. The value of this field shall be set to (Size of the Descriptor) - (Length of Descriptor Tag). When reading a descriptor the CRC should be validated.

2.2.2 Primary Volume Descriptor

```
struct PrimaryVolumeDescriptor {     /* ISO 13346 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 ISO/IEC 13346 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.

ISO 13346 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 ISO/IEC 13346 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 ISO/IEC 13346 (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 ISO/IEC 13346) 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

    ☞   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.

2.2.3 Anchor Volume Descriptor Pointer

```
struct AnchorVolumeDescriptorPointer {            /* ISO 13346 3/10.2 */
        struct tag            DescriptorTag;
        struct extent_ad      MainVolumeDescriptorSequenceExtent;
        struct extent_ad      ReserveVolumeDescriptorSequenceExtent;
        byte                  Reserved[480];
}
```

NOTE: An *AnchorVolumeDescriptorPointer* 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: Unclosed CD-R media may have an *Anchor Volume Descriptor Pointer* recorded at only sector 512. Upon close, CD-R media will conform to the rules above.

2.2.3.1 struct MainVolumeDescriptorSequenceExtent
The main *VolumeDescriptorSequenceExtent* shall have a minimum length of 16 logical sectors.

2.2.3.2 struct ReserveVolumeDescriptorSequenceExtent
The reserve *VolumeDescriptorSequenceExtent* shall have a minimum length of 16 logical sectors.

2.2.4 Logical Volume Descriptor

```
struct LogicalVolumeDescriptor {            /* ISO 13346 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 *LogicalVolumeDescriptor*.

✍   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 *LogicalVolumeDescriptor*. Since UDF requires that all Volumes within a VolumeSet 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:** 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 *DomainIdentifier* shall be set to:
> "***OSTA UDF Compliant**"

As described in the section on *Entity Identifier* the *IdentifierSuffix* 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 *IdentifierSuffix* field of this EntityID contains *SoftWriteProtect* and *HardWriteProtect* flags. Refer to 2.1.4.3.

2.2.4.4 **struct EntityID ImplementationIdentifier*;***
For more information on the proper handling of this field see the section on *Entity Identifier.*

2.2.4.5 **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.

2.2.4.6 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 and 2.2.9).

2.2.5 Unallocated Space Descriptor

```
struct UnallocatedSpaceDesc {                    /* ISO 13346 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.

2.2.6   Logical Volume Integrity Descriptor

```
struct LogicalVolumeIntegrityDesc { /* ISO 13346 3/10.10 */
        struct tag              DescriptorTag,
        Timestamp               RecordingDateAndTime,
        Uint32                  IntegrityType,
        struct extend_ad        NextIntegrityExtent,
        byte                    LogicalVolumeContentsUse[32],
        Uint32                  NumberOfPartitions,
        Uint32                  LengthOfImplementationUse,
        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?

20

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
See the section on *Logical Volume Header Descriptor* for information on the contents of this field.

2.2.6.2 Uint32 FreeSpaceTable
Since most operating systems require that an implementation provide the true free space of a Logical Volume at mount time it is important that these values be maintained. The optional value of #FFFFFFFF, which indicates that the amount of available free space is not known, shall not be used.

NOTE: The FreeSpaceTable 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. The optional value of #FFFFFFFF, which indicates that the partition size is not known, shall not be used.

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	??	Implementation Use	byte

Implementation ID - The implementation identifier *EntityID* of the implementation which last modified anything within the scope of this

21

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 associated Logical Volume. This information is needed by the Macintosh OS. All implementations shall maintain this information. NOTE: This value does not include Extended Attributes as part of the file count.

Number of Directories - The current number of directories in the associated Logical Volume. This information is needed by the Macintosh OS. All implementations shall maintain this information.
NOTE: The root directory shall be included in the directory count.

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 #0150 would indicate revision 1.50 of the UDF specification.

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 #0150 would indicate revision 1.50 of the UDF specification.

Maximum UDF Write Revision - Shall indicate the maximum revision of the UDF specification that an implementation which 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 #0150 would indicate revision 1.50 of the UDF specification.

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

2.2.7 Implemention Use Volume Descriptor

```
struct ImpUseVolumeDescriptor {
        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 which 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 Implementation Identifier
This field shall specify "*UDF LV Info".

2.2.7.2 bytes Implementation Use
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         ImplementionID,
        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
Identifies the Logical Volume referenced by this descriptor.

2.2.7.2.3 dstring LVInfo1

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 ImplementionID

Refer to the section 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 ISO 13346 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	#00 bytes
4	32	Partition Type Identifier	EntityID
36	2	Volume Sequence Number	Uint16
38	2	Partition Number	Uint16
40	24	Reserved	#00 bytes

- Partition Type Identifier:
    - Flags = 0
    - Identifier = *UDF Virtual Partition
    - IdentifierSuffix is recorded as in section 2.1.5.3
- Volume Sequence Number = volume upon which the VAT and Partition is recorded
- Partition Number = an identification of a partition within the volume identified by the volume sequence number

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. This 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	#00 bytes
4	32	Partition Type Identifier	EntityID
36	2	Volume Sequence Number	Uint16
38	2	Partition Number	Uint16
40	2	Packet Length	Uint16 = 32
42	1	Number of Sparing Tables (=N_ST)	Uint8
43	1	Reserved	#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
    - IdentifierSuffix is recorded as in section 2.1.5.3.
- 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. Shall be set to 32.
- 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 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) which 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 0. 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 0 and examining the contents for the EntityID at the end of the table.

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. Each sector can hold entries that represent up to 512 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 sector 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 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 be followed by a EntityID and a Uint32 entry indicating the location of the previous VAT ICB.

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.

27

Virtual Allocation Tablestructure

Offset	Name	Contents
0	LBA of virtual sector 0	Uint32
4	LBA of virtual sector 1	Uint32
8	LBA of virtual sector 2	Uint32
...	...	Uint32
2048	LBA of virtual sector 512	Uint32
...	...	Uint32
N * 4	Entity Identifier	EntityID
N * 4 + 32	Previous VAT ICB location	Uint32

An entry of #FFFFFFFF indicates that the virtual sector is currently unused.
The LBA specified is located in the partition identified by the partition map.
The number of entries in the table can be determined from the VAT file size in the ICB:

$$\text{Number of entries (N)} = \frac{FileSize - 36}{4}$$

The EntityID shall contain:

- Flags = 0
- Identifier = `*UDF Virtual Alloc Tbl`
- IdentifierSuffix is recorded as in UDF 2.1.5.3

2.2.11 Sparing Table

Certain disk/drive systems do not perform defect management (eg. CD-RW). 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 is 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. The sparing table further specifies an exception list of logical to physical

mappings. All mappings are one packet in length. The packet size is specified in the sparable partition map.

Available sparing areas may be anywhere on the media, either inside or outside of a partition. If located inside a partition, sparable space shall be marked as allocated and shall be included in the Non-Allocatable Space List. 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 0, indicating that the contents are not specified by ISO 13346.
- Sparing Identifier:
  - Flags = 0
  - Identifier = *UDF Sparing Table
  - IdentifierSuffix is recorded as in UDF 2.1.5.3
- 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 list.

2.3 Part 4 - File System
2.3.1 Descriptor Tag

```
struct tag {              /* ISO 13346 4/7.2 */
        Uint16       TagIdentifier;
        Uint16       DescriptorVersion;
        Uint8        TagChecksum;
        byte         Reserved;
        Uint16       TagSerialNumber;
        Uint16       DescriptorCRC;
        Uint16       DescriptorCRCLength;
        Uint32       TagLocation;
}
```

2.3.1.1 Uint16 TagSerialNumber

☞   Ignored.

✍   Reset to a unique value at volume initialization.

The *TagSerialNumber* shall be set to a value that differs from ones previously recorded, upon volume re-initialization. The intended use of this field is for disaster recovery. The *TagSerialNumber* for all descriptors in Part 4 should be the same as the serial number used in the associated File Set Descriptor

2.3.1.2 Uint16 DescriptorCRCLength

CRCs shall be supported and calculated for each descriptor, unless otherwise noted. The value of this field shall be set to: (Size of the Descriptor) - (Length of Descriptor Tag). When reading a descriptor the CRC should be validated.

2.3.2 File Set Descriptor

```
struct FileSetDescriptor {     /* ISO 13346 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              FileSetIdentifer[32];
        dstring              CopyrightFileIdentifier[32];
        dstring              AbstractFileIdentifier[32];
```

```
        struct long_ad          RootDirectoryICB;
        struct EntityID          DomainIdentifier;
        struct long_ad          NextExtent;
        byte                     Reserved[48];
}
```

Only one *FileSet* descriptor shall be recorded. On WORM media, multiple *FileSets* may be recorded.

The UDF provision for multiple File Sets is as follows:

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

Within a *FileSet* on WORM, if all files and directories have been recorded with ICB strategy type 4, then the *DomainID* of the corresponding *FileSet Descriptor* shall be marked as *HardWriteProtected*.

The intended purpose of multiple *FileSet*s on WORM is to support the ability to have multiple archive images on the media. For example one *FileSet* could represent a backup of a certain set of information made at a specific point in time. The next *FileSet* 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 ISO/IEC 13346 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 ISO/IEC 13346 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 ISO 13346 that are within the scope of the FileSetDescriptor.

    ✍   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 *ImplementationIdentifier* shall be set to:
               "***OSTA UDF Compliant**"

As described in the section on *Entity Identifier* the *IdentifierSuffix* 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 the section on *Entity Identifier.*

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

2.3.3 Partition Header Descriptor

```
struct PartitionHeaderDescriptor {     /* ISO 13346 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];
}
```

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.

NOTE: 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.

2.3.3.1 struct short_ad PartitionIntegrityTable

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

34

2.3.4 File Identifier Descriptor

```
struct FileIdentifierDescriptor {                    /* ISO 13346 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 one Logical Block.

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 Uint16 Lengthof ImplementationUse

☞   Shall specifiy the length of the *ImplementationUse* field.

✍   Shall specifiy the length of the *ImplementationUse* field. This field may be ZERO, indicating that the *ImplementationUse* field has not been used.

2.3.4.3 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 the section on *Entity Identifier*.

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

2.3.5 ICB Tag

```
struct icbtag { /* ISO 13346 4/14.6 */
        Uint32          PriorRecordedNumberofDirectEntries;
        Uint16          StrategyType;
        byte            StrategyParameter[2];
        Uint16          NumberofEntries;
        byte            Reserved;
        Uint8           FileType;
        Lb_addr         ParentICBLocation;
        Uint16          Flags;
}
```

2.3.5.1 Uint16 StrategyType

   ✐   The contents of this field specifies the ICB strategy type used. For the purposes of read access an implementation shall support strategy types 4 and 4096.

   ✍   Shall be set to 4 or 4096.

NOTE: Strategy type 4096, which is defined in the appendix, is intended for primary use on WORM media, but may also be used on rewritable and overwritable media.

2.3.5.2 Uint8 FileType

As a point of clarification a value of 5 shall be used for a standard byte addressable file, *not 0.*

2.3.5.3 ParentICBLocation

The use of this field by is optional.
NOTE: In ISO 13346-4/14.6.7 it states that "If this field contains 0, then no such ICB is specified." This is a flaw in the ISO 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 the section 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 may indicate (ONE) that the file is non-relocatable. An implementation may reset this bit to ZERO to indicate that the file is relocatable if the implementation can not assure that the file will not be relocated.

✍ Should be 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 {               /* ISO 13346 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 *FileEntry* shall not exceed the size of one logical block.

2.3.6.1 Uint8 RecordFormat;

 ☞ For OSTA UDF compliant media this bit shall indicate (ZERO) 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 this bit shall indicate (ZERO) 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 Uint8 RecordLength;

✍ For OSTA UDF compliant media this bit shall indicate (ZERO) 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 struct EntityID ImplementationIdentifier;

Refer to the section on *Entity Identifier*.

2.3.6.5 Uint64 UniqueID

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

It is required that this value be maintained and unique for every file and directory in the LogicalVolume. This includes FileEntry descriptors defined for Extended Attribute spaces. The FileEntry for the Extended Attribute space shall contain the same *UniqueID* as the file to which it is attached.

NOTE: The *UniqueID* values 1-15 shall be reserved for the use of Macintosh implementations.

2.3.7 Unallocated Space Entry

```
struct UnallocatedSpaceEntry {                    /* ISO 13346 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 (ISO 13346 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 {            /* ISO 13346 4/14.11 */
        struct Tag      DescriptorTag;
        Uint32          NumberOfBits;
        Uint32          NumberOfBytes;
        byte            Bitmap[??];
}
```

2.3.8.1 struct Tag DescriptorTag

The calculation and maintenance of the *DescriptorCRC* field of the Descriptor Tag for the *SpaceBitmap* descriptor is optional. If the CRC is not maintained then both the *DescriptorCRC* and *DescriptorCRCLength* fields shall be ZERO.

2.3.9   Partition Integrity Entry

```
struct PartitionIntegrityEntry {                /* ISO 13346 4/14.13 */
        struct tag              DescriptorTag;
        struct icbtag           ICBTag;
        struct timestamp        RecordingTime;
        Uint8                   IntegrityType;
        byte                    Reserved[175];
        struct EntityID         ImplementationIdentifier;
        byte                    ImplementationUse[256];
}
```

With the functionality of the *Logical Volume Integrity Descriptor* this descriptor is not needed, 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 stand alone drive.

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

Long Allocation Descriptor - For a Logical Volume that resides on a single Logical 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: 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.

2.3.10.1 Long Allocation Descriptor

```
struct long_ad {                 /* ISO 13346 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 { /* ISO 13346 4/14.5 */

```
            struct tag        DescriptorTag;
            Uint32            PreviousAllocationExtentLocation;
            Uint32            LengthOfAllocationDescriptors;
    }
```

NOTE:. *AllocationDescriptor* extents shall be a maximum of one logical block in
length.

2.3.11.1 Uint12 PreviousAllocationExtentLocation
    ☞   The previous allocation extent location shall not be used as specified
        below.

    ✍   Shall be set to 0.

2.3.12   Pathname
2.3.12.1   Path Component
```
    struct PathComponent {                    /* ISO 13346 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.

2.3.13 Non-Allocatable Space List
ISO 13346 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
List* provides a method to describe space not usable by the file system. The *Non-
Allocatable Space List* shall be recorded only on media systems that do not do defect
management (eg. CD-RW).

The *Non-Allocatable Space List* shall be generated at format time. All space indicated by
the *Non-Allocatable Space List* shall also be marked as allocated in the free space map.
The *Non-Allocatable Space List* shall be recorded as a file of the root directory. The file
name "Non-Allocatable Space" (#4E, #6F, #6E, #2D, #41, #6C,
#6C,#6F,#61,#74,#61,#62,#6C,#65, #20, #70, #61, #63, #65) shall be used. The file shall

be marked with the attributes Hidden (bit 0 of file characteristics set to ONE) and System (bit 10 of ICB flags field set to ONE). The name may be recorded in any legal word size. The information length of this file shall be zero. This file shall have all Non-Allocatable sectors identified by its allocation extents. The allocation extents shall indicate that each extent is allocated but not recorded. This list shall include both defective sectors found at format time and space allocated for sparing at format time.

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 {       /* ISO 13346 1/7.3 */
        Uint16          TypeAndTimezone;
        Uint16          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 {   /* ISO 13346 4/14.15 */
        Uint64          UniqueID,
        bytes           reserved[24]
}
```

3.2.1.1 Uint64 UniqueID

This field contains the next *UniqueID* value which should be used.

NOTE: For compatibility with Macintosh systems implementations should keep this value less than the maximum value of a Int32 (2^{31} - 1).

3.3 Part 4 - File System
3.3.1 File Identifier Descriptor

```
struct FileIdentifierDescriptor {                /* ISO 13346 4/14.4 */
        struct tag      DescriptorTag;
        Uint16          FileVersionNumber;
        Uint8           FileCharacteristics;
        Uint8           LengthofFileIdentifier;
        struct long_ad ICB;
        Uint16          LengthofImplementationUse;
        byte            ImplementationUse[??];
        char            FileIdentifier[??];
        byte            Padding[??];
}
```

NOTE: 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 Root, as stated in ISO 13346-4, section 8.6

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 *FileIdentifier* structure 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
Under UNIX 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 { /* ISO 13346 4/14.6 */
        Uint32          PriorRecordedNumberofDirectEntries;
        Uint16          StrategyType;
        byte            StrategyParameter[2];
        Uint16          NumberofEntries;
        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).

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).

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.

3.3.3 File Entry

```
struct FileEntry {                /* ISO 13346 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 *FileEntry* 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 which 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?

User, 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.

Processing Permissions

Implementation shall process the permission bits according to the following table which 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
Read	file	The file may be read	E	E	E	E	E	E
Read	directory	The directory may be read	E	E	E	E	E	E
Write	file	The file's contents may be modified	E	E	E	E	E	E
Write	directory	Files or subdirectories may be created, deleted or renamed	E	E	E	E	E	E
Execute	file	The file by be executed.	I	I	I	I	I	E
Execute	directory	The directory may be searched for a specific file or subdirectory.	E	E	E	E	E	E
Attribute	file	The file's permissions may be changed.	E	E	E	E	E	E
Attribute	directory	The directory's permissions may be changed.	E	E	E	E	E	E
Delete	file	The file may be deleted.	E	E	E	E	E	E
Delete	directory	The directory may be deleted.	E	E	E	E	E	E

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.

NOTE: 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.

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
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 by 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.

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.

3.3.3.4   Uint64 UniqueID

NOTE: For some operating systems (i.e. Macintosh) this value needs to be less than the max value of a *Int32* (2^{31} - 1). Under the Macintosh operating system this value is used to represent the Macintosh directory/file ID. Therefore an implementation should attempt to keep this value less than the max value of a *Int32* (2^{31} - 1). The values 1-15 shall be reserved for the use of Macintosh implementations.

3.3.3.5   byte Extended Attributes

Certain extended attributes should be recorded in this field of the *FileEntry* for performance reasons. Other extended attributes should be recorded in an ICB pointed to by the field *ExtendedAttributeICB*. In the section 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) which 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.
2. Smaller EAs shall be constrained to an attribute length which is a multiple of 4 bytes.
3. The Extended Attribute space shall appear as a single contiguous logical space constructed as follows:

ISO/IEC 13346 EAs
Non block aligned Implementation Use EAs
Block aligned Implementation Use EAs
Application Use EAs

3.3.4.1 Extended Attribute Header Descriptor

```
struct ExtendedAttributeHeaderDescriptor {  /* ISO 13346 4/14.10.1 */
        struct tag      DescriptorTag;
        Uint32          ImplementationAttributesLocation;
        Uint32          ApplicationAttributesLocation;
}
```

If the attributes associated with the *location* fields highlighted above do not exist, then the value of the *location* field shall point to the byte after the extended attribute space.

3.3.4.2 Alternate Permissions

```
struct AlternatePermissionsExtendedAttribute {       /* ISO 13346 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 {  /* ISO 13346 4/14.10.5 */
        Uint32          AttributeType;
        Uint8           AttributeSubtype;
        byte            Reserved[3];
        Uint32          AttributeLength;
        Uint32          DataLength;
        Uint32          FileTimeExistence;
        byte            FileTimes;
}
```

3.3.4.3.1 Uint32 FileTimeExistance
3.3.4.3.1.1 Macintosh OS
> This field shall be set to indicate that only the file creation time has been
> recorded.

3.3.4.3.1.2 Other OS
> This structure need not be recorded.

3.3.4.3.2 byte FileTimes
3.3.4.3.2.1 Macintosh OS
> ☞   Shall be interpreted as the creation time of the associated file.
>
> ✍   Shall be set to creation time of the associated file.
>
> If the *File Times Extended Attribute* does not exist then a Macintosh
> implementation shall use the *ModificationTime* field of the *File Entry* to
> represent the file creation time.

3.3.4.3.2.2 Other OS
> This structure need not be recorded.

3.3.4.4 Device Specification Extended Attribute
```
struct DeviceSpecificationExtendedAttribute {       /* ISO 13346 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 *FileType* field in the *icbtag* structure be set to 6 (indicating a block special device file), OR 7 (indicating a character special device file).

> If the contents of the *FileType* 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 *FileType* field in the *icbtag* structure equal 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.

> All implementations shall record a developer ID in the *ImplementationUse* field that uniquely identifies the current implementation.

3.3.4.5 Implementation Use Extended Attribute

```
struct ImplementationUseExtendedAttribute {        /* ISO 13346 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 attribute space. C source code for the header checksum may be found in the appendix.

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 attribute 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-1	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 attribute space. The *FreeEASpace* extended attribute may be overwritten and the space re-used by any implementation who 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**"

56

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 Consortium (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 supported through the use of the following two *Implementation Use Extended Attributes.*

3.3.4.5.3.1 OS2EA
This extended attribute contains all OS/2 definable extended attributes which shall be stored as an *Implementation Use Extended Attribute* whose *ImplementationIdentifier* shall be set to:
"***UDF OS/2 EA**"

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

OS2EA format

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	IU_L-2	OS/2 Extended Attributes	FEA

The *OS2ExtendedAttributes* field 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*

3.3.4.5.3.2 OS2EALength

This attribute specifies the OS/2 Extended Attribute 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 *FileEntry*. 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 *ImplementationUseLength* field of the **OS2EA** extended
attribute - 2.

3.3.4.5.4 Macintosh OS

The Macintosh OS requires the use of the following four 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 *FileEntry*.

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 *FileEntry*.

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 (=0)	Uint16
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 (=0)	Uint16
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 there 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 which are in *big endian* format.

3.3.4.5.4.3 MacUniqueIDTable

This extended attribute contains a table used to look up the *FileEntry* for a specified *UniqueID*. This table shall be stored as an *Implementation Use Extended Attribute* whose *ImplementationIdentifier* shall be set to:
>   "***UDF Mac UniqueIDTable**"

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

MacUniqueIDTable format

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	2	Reserved for padding (=0)	Uint16
4	4	Number of Unique ID Maps (=N_DID)	Uint32
8	N_DID x 8	Unique ID Maps	UniqueIDMap

UniqueIDMap format

RBP	Length	Name	Contents
0	8	File Entry Location	small_ad

small_ad format

RBP	Length	Name	Contents
0	2	Extent Length	Uint16
2	6	Extent Location	lb_addr (4/7.1)

This *UniqueIDTable* is used to look up the corresponding *FileEntry* for a specified Macintosh directory/file ID (*UniqueID*). For example, given some Macintosh directory/file ID *i* the corresponding *FileEntry* location may be found in the *(i-2) UniqueIDMap* in the *UniqueIDTable*. The correspondence of directory/file ID to *UniqueID* is (*Directory/file ID -2)* because Macintosh directory/file IDs start at 2 while *UniqueID*s start at 0. In the Macintosh the root directory always has a directory ID of 2, which corresponds to the requirement of having the *UniqueID* of the root *FileEntry* have the value of 0.

If the value of the *Extent Length* field of the *File Entry Location* is 0 then the corresponding *UniqueID* is free.

The *MacUniqueIDTable* extended attribute shall be recorded as an extended attribute of the root directory.

The *MacUniqueIDTable* is created and updated only by implementations that support the Macintosh. When the Logical Volume is modified by implementations that do not support the *MacUniqueIDTable* can become out of date in the following ways:
- Files can exist on the media which are not referenced in the *MacUniqueIDTable*. This can result from a non-Macintosh implementation creating a new file on the media.
- Files in the UniqueID table may no longer exist on the media. This can result from a non-Macintosh implementation deleting a file on the media

The Macintosh uses the UniqueID to directly address a file on the media without reference to its file name. This will only happen if the file was originally created by an implementation that supports the Macintosh. Therefore any new files added to the logical volume by non-Macintosh implementations will always be referenced by file name first, never by UniqueID. At the first access of the file by file name, the Macintosh implementation can detect that this UniqueID is not in the *MacUniqueIDTable* and update the table appropriately.

The second problem is a little more difficult to address. The problem occurs when a Macintosh implementation gets a reference to a file on the media given a UniqueID. The Macintosh implementation needs to make sure that the file the UniqueID references still exists. The following things can be done:

- Verify that the File Entry (FE) pointed to by the UniqueID contains the same UniqueID.
- AND Verify that the block that contains the FE is not on the free list. This could occur when the file is deleted by a non-Macintosh implementation, and the FE has not been overwritten.

The only case that these two tests do not catch is when a file has been deleted by a non-Macintosh implementation, and the logical block associated with the FE has been reassigned to a new file, and the new file has used the block in an extent of *Allocated but not recorded.*

3.3.4.5.4.4 MacResourceFork

This extended attribute contains the Macintosh resource fork data for the associated file.   The resource fork data shall be stored as an *Implementation Use Extended Attribute* whose *ImplementationIdentifier* shall be set to:

"***UDF Mac ResourceFork**"

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

MacResourceFork format

RBP	Length	Name	Contents
0	2	HeaderChecksum	Uint16
2	IU_L-2	Resource Fork Data	bytes

The *MacResourceFork* extended attribute shall be recorded as an extended attribute of all files, with > 0 bytes in the resource fork, within the Logical Volume.

The two fields of the *MacFinderInfo* extended attribute the reference the *MacResourceFork* extended attributes are defined as follows:

Resource Fork Data Length - Shall be set to the length of the actual data considered to be part of the resource fork.
Resource Fork Allocated Length - Shall be set to the total amount of space in bytes allocated to the resource fork.

3.3.4.5.5 UNIX

&⁓   Ignored.

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

3.3.4.6 Application Use Extended Attribute

```
struct ApplicationUseExtendedAttribute {     /* ISO 13346 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 contains 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 attribute space. C source code for the header checksum may be found in the appendix.

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

This extended attribute shall be used to indicate unused space within the extended attribute 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:

<div align="center"><i>FreeAppEASpace</i> format</div>

RBP	Length	Name	Contents
0	2	Header Checksum	Uint16
2	IU_L-1	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 attribute space. The *FreeAppEASpace* extended attribute may be overwritten and the space re-used by any implementation who sees a need to overwrite it.

4. User Interface Requirements

4.1 Part 3 - Volume Structure

Part 3 of ISO/IEC 13346 contains various Identifiers which, depending upon the implementation, may have to be presented to the user.

- *VolumeIdentifier*
- *VolumeSetIdentifier*
- *LogicalVolumeID*

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 4.1.2.1.

C source code for the translation algorithms may be found in the appendices of this document.

4.2 Part 4 - File System

4.2.1 ICB Tag

```
struct icbtag { /* ISO 13346 4/14.6 */
        Uint32          PriorRecordedNumberofDirectEntries;
        Uint16          StrategyType;
        byte            StrategyParameter[2];
        Uint16          NumberofEntries;
        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 :

> *FileType* 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 (*SymbolicLink*) shall access the file/directory to which the symbolic link is pointing.

4.2.2 File Identifier Descriptor

```
struct FileIdentifierDescriptor {                /* ISO 13346 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 which 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 the appendices of this document.

NOTE: In the definition of the following algorithms anytime a d-character is specified in quotes, the Unicode hexadecimal value will also be specified. In addition 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.

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).

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:** 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 :

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 *FileIdentifer*: 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 the appendix 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 a 4 digit CS0 Hex representation of the 16-bit CRC of the original CS0 *FileIdentifier*. **NOTE:** All other algorithms *except DOS* precede the CRC by a separator '#' (#0023). Due to the limited number of characters in a DOS file name a separator for the CRC is not used.

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-insensitive comparison shall be performed.

2. Validate *FileIdentifer*: 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 the appendix 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-insensitive comparison shall be performed.
2. Validate *FileIdentifer*: 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 the appendix 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-insensitive comparison shall be performed.
2. Validate *FileIdentifer*: 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 the appendix 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 *FileIdentifer*: 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 the appendix 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 of the 16-bit CRC of the original CS0 *FileIdentifier.*

5. Informative

5.1 Descriptor Lengths

The following table summarizes the UDF limitations on the lengths of the Descriptors described in ISO 13346.

Descriptor	Length
Anchor Volume Descriptor Pointer	512
Volume Descriptor Pointer	512
Implementation Use 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
Unallocated Space Entry	Maximum of a Logical Block Size
Space Bit Map Descriptor	no max
Partition Integrity Entry	N/A

5.2 Using Implementation Use Areas
5.2.1 Entity Identifiers
Refer to the section 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 it may be reallocated by some type of logical volume repair facility. Orphan space is defined as space that is not directly or indirectly referenced by any of the non-implementation use descriptors defined in ISO 13346.

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

5.3 Boot Descriptor

Please refer to the "OSTA Native Implementation Specification" document for information on the Boot Descriptor.

5.4 Technical Contacts

Technical questions regarding this document may be emailed to the *OSTA Technical Committee* at **info@osta.org**. Also technical questions may be faxed to the attention of the *OSTA Technical Committee* at 1-805-962-1542.

OSTA may also be contacted through the following address:

    Technical Committee Chairman
    OSTA
    311 East Carrillo Street
    Santa Barbara, CA 93101
    (805) 963-3853

Also monitor the OSTA web site at **www.osta.org** for additional information.

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 complaint 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 attribute space.
"*UDF FreeAppEASpace"	Contains free unused space within the application extended attribute space.
"*UDF DVD CGMS Info"	Contains DVD Copyright Management Information
"*UDF OS/2 EA"	Contains OS/2 extended attribute data.
"*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 Mac UniqueIDTable"	Contains Macintosh UniqueID Table which is used to map a Unique ID to a File Entry.
"*UDF Mac ResourceFork"	Contains Macintosh resource fork information.
"*UDF Virtual Partition"	Describes UDF Virtual Partition
"*UDF Sparable Partition"	Describes UDF Sparable Partition
"*UDF Virtual Alloc Tbl"	Contains information for handling rewriting to sequentially written media.
"*UDF Sparing Table"	Contains information for handling defective areas on the media

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 EA"	#2A, #55, #44, #46, #41, #20, #45, #41
"*UDF OS/2 EALength"	#2A, #55, #44, #46, #20, #45, #41, #4C, #65, #6E, #67, #74, #68
"*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 Mac UniqueIDTable"	#2A, #55, #44, #46, #20, #4D, #61, #63, #20, #55, #6E, #69, #71, #75, #65, #49, #44, #54, #61, #62, #6C, #65
"*UDF Mac ResourceFork"	#2A, #55, #44, #46, #20, #4D, #61, #63, #20, #52, #65, #73, #6F, #75, #72, #63, #65, #46, #6F, #72, #6B
"*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 Virtual Alloc Tbl"	#2A, #55, #44, #46, #20, #56, #69, #72, #74, #75, #61, #6C, #20, #41, #6C, #6C, #6F, #63, #20, #54, #62, #6C
"*UDF Sparing Table"	#2A, #55, #44, #46, #20, #53, #70, #61, #72, #69, #6E, #67, #20, #54, #61, #62, #6C, #65

6.3 Operating System Identifiers

The following tables define the current allowable values for the *OS Class* and *OS Identifier* fields in the *IdentifierSuffix* of Entity Identifiers.

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
7-255	Reserved

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 System 7
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
5	0	Windows 95
6	0	Windows NT

For the most update list of values for OS Class and OS Identifier please contact OSTA and request a copy of the *UDF Entity Identifier Directory*. This directory will also contain Implementation Identifiers of ISVs who have provided the necessary information to OSTA.

NOTE: If you wish to add to this list please contact the OSTA Technical Committee Chairman at the OSTA address listed in section *5.3 Technical Contacts*. ***Currently not all***

features of Windows NT and NetWare are fully supported by this specification, but OSTA has started the work on these operating systems.

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)
         {
```

80

```
                /*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 ISO/IEC 13346.

```c
/*
 *      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;
}

#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);
```

82

```
        exit(0);
}
#endif
```

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

```c
#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.
Copyright is held by AT&T.

AT&T gives permission for the free use of the above source code.

6.6 Algorithm for Strategy Type 4096

This section describes a strategy for constructing an ICB hierarchy. For 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:
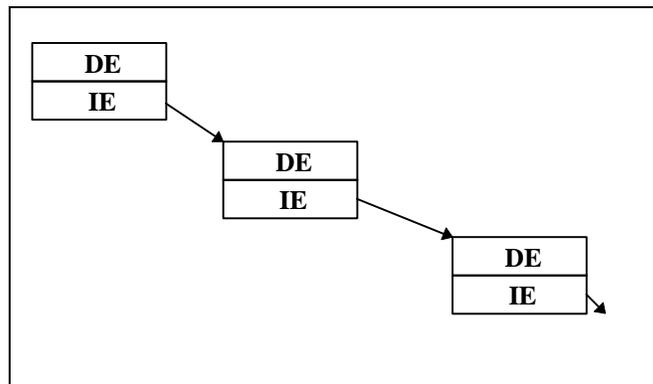
NOTE: This strategy builds an ICB hierarchy that is a simple linked list of direct entries.

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*, *LogicalVolumeID and FileSetID*.

6.7.1 DOS Algorithm

```
/***********************************************************************
 * OSTA UDF compliant file name translation routine for DOS.
 * 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>

#define DOS_NAME_LEN     8
#define DOS_EXT_LEN      3
#define ILLEGAL_CHAR_MARK 0x005F
#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 short unicode_t;
typedef unsigned char byte;

/*** PROTOTYPES ***/
unsigned short cksum(register unsigned char *s, register int n);
int IsIllegal(unicode_t current);

/* Define functions or macros to both determine if a character
 * is printable and compute the uppercase version of a character
 * under your implementation.
 */
int UnicodeIsPrint(unicode_t);
unicode_t UnicodeToUpper(unicode_t);

/***********************************************************************
 * Translate udfName to dosName using OSTA compliant.
 * dosName must be a unicode string with min length of 12.
 *
 * RETURN VALUE
 *    Number of unicode characters in dosName.
 */
int UDFDOSName(
unicode_t *dosName,  /* (Output)DOS compatible name.   */
unicode_t *udfName,  /* (Input) Name from UDF volume.  */
int        udfLen,   /* (Input) Length of UDF Name.    */
byte      *fidName,  /* (Input) Bytes as read from media */
int        fidNameLen)/* (Input) Number of bytes in fidName.*/
{
    int index, dosIndex = 0, extIndex = 0, lastPeriodIndex;
```

```c
    int needsCRC = FALSE, hasExt = FALSE, writingExt = FALSE;
    unsigned short valueCRC;
    unicode_t ext[DOS_EXT_LEN], current;

    /*Used to convert hex digits. Used ASCII for readability. */
    const char hexChar[] = "0123456789ABCDEF";

    for (index = 0 ; index < udfLen ; index++)
    {
        current = udfName[index];
        current = UnicodeToUpper(current);

        if (current == PERIOD)
        {
            if (dosIndex==0 || hasExt)
            {
                /* Ignore leading periods or any other than
                 * used for extension.
                 */
                needsCRC = TRUE;
            }
            else
            {
                /* First, find last character which is NOT a period
                 * or space.
                 */
                lastPeriodIndex = udfLen - 1;
                while(lastPeriodIndex >=0 &&
                        (udfName[lastPeriodIndex]== PERIOD ||
                         udfName[lastPeriodIndex] == SPACE))
                {
                    lastPeriodIndex--;
                }

                /* Now search for last remaining period. */
                while(lastPeriodIndex >= 0 &&
                        udfName[lastPeriodIndex] != PERIOD)
                {
                    lastPeriodIndex--;
                }

                /* See if the period we found was the last or not. */
                if (lastPeriodIndex != index)
                {
                    needsCRC = TRUE; /* If not, name needs translation. */
                }

                /* As long as the period was not trailing,
                 * the file name has an extension.
                 */
                if (lastPeriodIndex >= 0)
                {
                    hasExt = TRUE;
                }
            }
        }
        else
        {

            if ((!hasExt && dosIndex == DOS_NAME_LEN) ||
                 extIndex == DOS_EXT_LEN)
            {
                /* File name or extension is too long for DOS. */
                needsCRC = TRUE;
            }
            else
            {
                if (current == SPACE)   /* Ignore spaces. */
```

87

```c
            {
                needsCRC = TRUE;
            }
            else
            {
                /* Look for illegal or unprintable characters. */
                if (IsIllegal(current) || !UnicodeIsPrint(current))
                {
                    needsCRC = TRUE;
                    current = ILLEGAL_CHAR_MARK;
                    /* Skip Illegal characters(even spaces),
                     * but not periods.
                     */
                    while(index+1 < udfLen
                            && (IsIllegal(udfName[index+1])
                            || !UnicodeIsPrint(udfName[index+1]))
                            && udfName[index+1] != PERIOD)
                    {
                        index++;
                    }
                }

                /* Add current char to either file name or ext. */
                if (writingExt)
                {
                    ext[extIndex++] = current;
                }
                else
                {
                    dosName[dosIndex++] = current;
                }
            }
        }
    }
    /* See if we are done with file name, either because we reached
     * the end of the file name length, or the final period.
     */
    if (!writingExt && hasExt && (dosIndex == DOS_NAME_LEN ||
                    index == lastPeriodIndex))
    {
        /* If so, and the name has an extension, start reading it. */
        writingExt = TRUE;
        /* Extension starts after last period. */
        index = lastPeriodIndex;
    }
}

/*Now handle CRC if needed. */
if (needsCRC)
{
    /* Add CRC to end of file name or at position 4. */
    if (dosIndex >4)
    {
        dosIndex = 4;
    }

    valueCRC = cksum(fidName, fidNameLen);

    /* Convert 16-bit CRC to hex characters. */
    dosName[dosIndex++] = hexChar[(valueCRC & 0xf000) >> 12]
    dosName[dosIndex++] = hexChar[(valueCRC & 0x0f00) >> 8];
    dosName[dosIndex++] = hexChar[(valueCRC & 0x00f0) >> 4];
    dosName[dosIndex++] = hexChar[(valueCRC & 0x000f)];
}

/* Add extension, if any. */
if (extIndex != 0)
{
```

```
          dosName[dosIndex++] = PERIOD;
          for (index = 0; index < extIndex; index++)
          {
              dosName[dosIndex++] = ext[index];
          }
      }

      return(dosIndex);
}

/***********************************************************************
 * Decides if a Unicode character matches one of a list
 * of ASCII characters.
 * Used by DOS 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);
}

/***********************************************************************
 * Decides whether character passed is an illegal character for a
 * DOS file name.
 *
 * RETURN VALUE
 *
 *    Non-zero if file character is illegal.
 */
int IsIllegal(
unicode_t ch)  /* (Input) character to test. */
{
    /* Genuine illegal char's for DOS. */
    if (ch < 0x20 || UnicodeInString("\\/:*?\"<>|", ch))
    {
        return(1);
    }
    else
    {
        return(0);
    }
}
```

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 cksum(unsigned char *s, 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.   */
byte *fidName,      /* (Input)  Bytes as read from media. */
int fidNameLen)     /* (Input)  Number of bytes in fidName. */
{
    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) || !isprint(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])
                               || !isprint(udfName[extIndex + index + 2])))
                    {
                        index++;
                    }
                }
                ext[localExtIndex++] = current;
            }

            /* Truncate filename to leave room for extension and CRC. */
            maxFilenameLen = ((MAXLEN - 4) - 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 = cksum(fidName, fidNameLen);
        /* 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)
```

93

```
    {
        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 Checksum Algorithm

```
/*
 * Calculates a 16-bit checksum of the Implementation Use
 * Extended Attribute header.  The fields AttributeType
 * through ImplementationIdentifier 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 6.11. For planned operating system, check the *Other* box and write in DVD.

6.9.1 Constraints imposed by UDF for 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 ISO 13346 and UDF. This will ease playing of DVD-Video in computer systems. Examples of such data include the time, date, permission bits, and a free space map (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.

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 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: The *DVD Specifications for Read-Only Disc* is a document, developed by the DVD Consortium, that 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 which 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 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 a ISO 13346 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 checksum in byte 4 and 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 can not 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 logical block number.

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:

> Toshiba Corporation
> Toshiba BLDG. 13D
> DVD Division
> 1-1 Shibaura 1-Chome, Minato-ku, Tokyo 105-01, JAPAN
> Mr. Y. Mizutani
> E-mail: 000092030295@tg-mail.toshiba.co.jp

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.

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).

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

ISO 13346 requires an Anchor Volume Descriptor Pointer (AVDP) at sector 256 and either N or (N - 256), where n is the last recorded Physical Address on the media. UDF requires that the AVDP be recorded at both sector 256 and sector (N - 256) when each session is closed. The file system may be in an intermediate state before closing and still be interchangeable, but not strictly in compliance with ISO 13346. In the intermediate state, only one AVDP exists. It should exist at sector 256, but if this is not possible due to a track reservation, it shall exist at sector 512.

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. Computer based implemenations are expected to handle VAT sizes of at least 64K bytes; dedicated player implementations may handle only smaller sizes.

6.10.1.1   Requirements

- 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.

- 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

- An intermediate state is allowed on CD-R media in which only one AVDP is recorded; this single AVDP shall be at sector 256 or sector 512 and according to the multisession rules below.

- Sequential file system writing shall be performed with variable packet writing. This allows maximum space efficiency for large and small updates. Variable packet writing is more compatible with CD-ROM drives as current models do not support method 2 addressing required by fixed packets.

- 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 Physical Address. If the VAT ICB is present, the volume is clean; otherwise it is dirty.

- The Partition Header descriptor, if recorded, shall specify no Unallocated Space Table, no Unallocated Space Bitmap, no Partition Integrity Table, 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. CD media should contain 1 write once partition and 1 virtual partition.

6.10.1.2 "Bridge" formats

ISO 9660 requires a Primary Volume Descriptor (PVD) at sector 16. If an ISO 9660 file system is desired, it may contain references to the same files as those referenced by ISO 13346 structures, or reference a different set of files, or a combination of the two.

It is assumed that early implementations will record some ISO 9660 structures but that as implementations of UDF become available, the need for ISO 9660 structures will decrease.

If an ISO 9660 bridge disc contains Mode 2 Form 1 sectors, then the CD-ROM XA extensions of ISO 9660 must be used.

6.10.1.3 End of session data

A session is closed to enable reading by CD-ROM drives. The last complete session on the disc shall conform completely to ISO 13346 and have two AVDPs recorded. This shall be accomplished by writing data according to End of session data table below. Although not shown in the following example, the data may be written in multiple packets.

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 ISO 13346.

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.

- 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).

- The host shall maintain a list of defects on the disc using a Non-Allocatable Space List (see 2.3.13).

- 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. This physical format may be followed by a verification pass. Defects found during the verification pass shall be enumerated in the *Non-Allocatable Space* list (2.3.13). 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 Physical Address 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 previously recorded packet.

- Update the sparing table to reflect any new spare areas

- Adjust the partition map as appropriate

- Update the free space map 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. The start of the partition shall be on a packet boundary. The partition length shall be an integral multiple of the packet size.

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
The Volume Recognition Sequence and Anchor Volume Descriptor Pointer locations are specified by ISO 13346 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 Physical Address of the first data sector in the first recorded data track in the last existent session of the volume. 'S' is the same value currently used in multisession ISO 9660 recording. The first track in the session shall be a data track.

'N' is the physical sector number of the last recorded data sector on a disc.
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. 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.

6.10.3.1 Volume Recognition Sequence
The following descriptions are added to UDF (see also ISO/IEC 13346 Part 2) in order to handle a multisession disc.

- The volume recognition area of the UDF Bridge format shall be the part of the volume space starting at sector $S + 16$.
- The volume recognition space shall end in the track in which it begins. As a result of this definition, the volume recognition area always exists in the last session of a disc.
- When recorded in Random Access mode, a duplicate Volume Recognition Sequence shall be recorded beginning at sector N - 256.

6.10.3.2 Anchor Volume Descriptor Pointer
Anchor Volume Descriptor Pointers (AVDP) shall be recorded at the following logical sector numbers: $S + 256$ and N - 256. The AVDP at sector N - 256 shall be recorded before closing a session; it may not be recorded while a session is open.

6.10.3.3 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 "Multisession and Mixed Mode" section above. The disc may contain sessions that are based on ISO 9660, audio, vendor unique, or a combination of file systems. The UDF Bridge format allows CD enhanced discs to be created.

The UDF session may contain pointers to data in other sessions, pointers to data only within the UDF session, or a combination of both. Some examples of UDF Bridge discs are shown below.

Multisession UDF disc

Access to LSN=16+x   Access to LSN=256

16 sectors

256 sectors

N - 256

16 sectors

256 sectors

LSN=0

LSN=*S*

First Session

1st Recorded Track in the last session

: Volume recognition area

: Anchor point

CD enhanced disc

1st session

2nd session

UDF Session

Playable by conventional CD-Player

Used by UDF
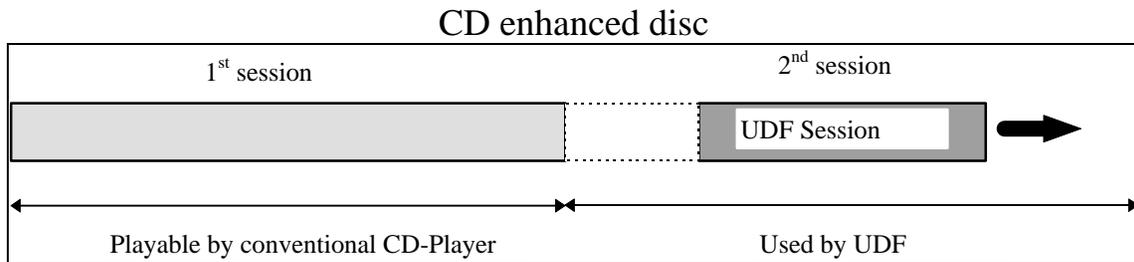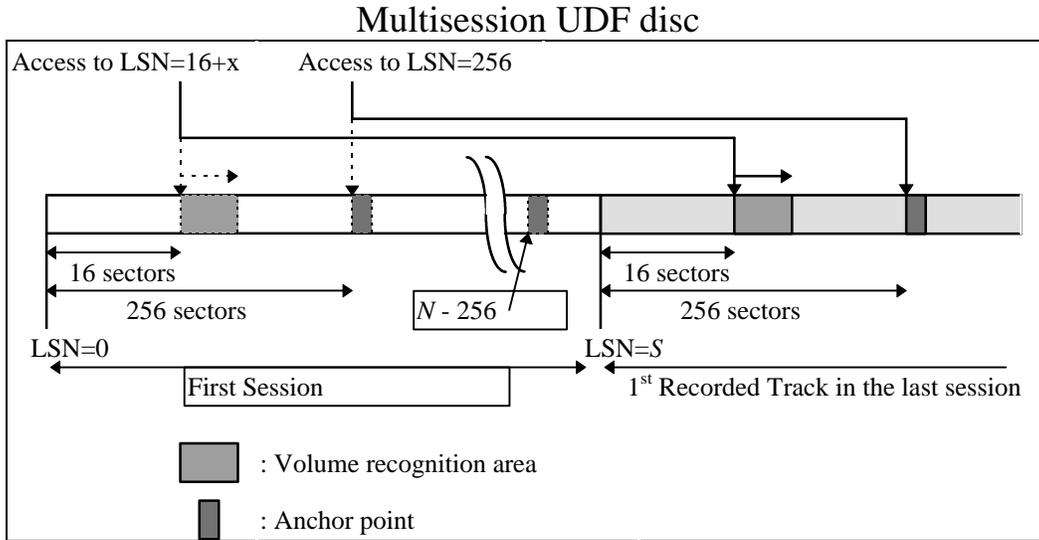
ISO 9660 converted to UDF

1st session	2nd session		3rd session
9660 Session	9660 Session		UDF Session

Written by conventional 9660 formatter software

Managed by UDF
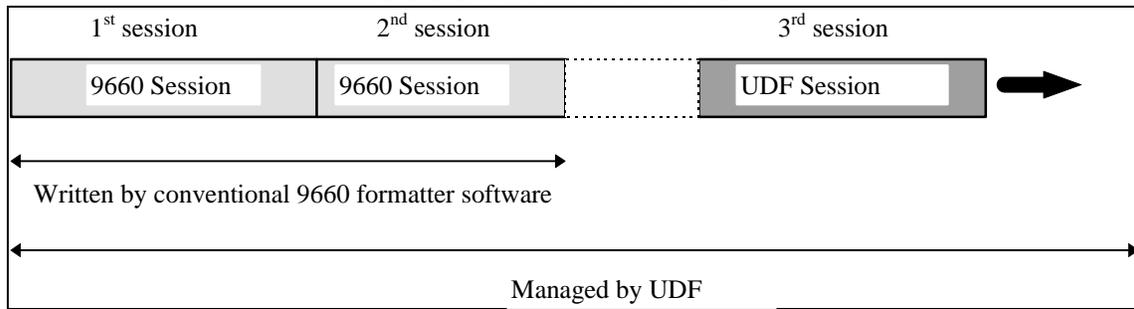
Foreign format converted to UDF

1st session	2nd session		3rd session
Data Session	Data Session		UDF Session

Written by another file system

Managed by UDF
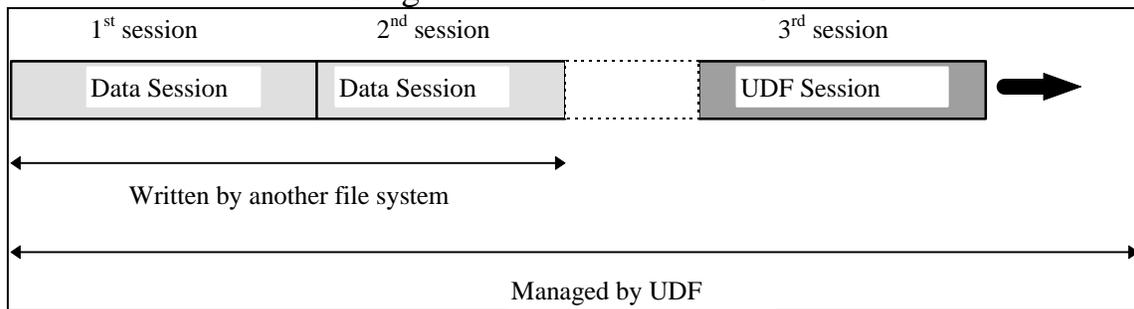
6.11 UDF Media Format Revision History

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 DCN 2-015.

Description	DCN	Updated in UDF Revision	Minimum UDF Read Revision	Minimum UDF Write Revision
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
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	1.50	1.50
Reccommmendations for CD Media	2-031	1.50	1.50	1.50

6.12 Developer Registration Form

Any developer that plans on implementing ISO/IEC 13346 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 *Implementation Identifiers*. The developers on this list are encouraged to interchange media to verify data interchange among implementations.
- Notification of OSTA Technical Committee meetings. You may attend a limited number of this meetings without becoming an official OSTA member.
- You can be added to the OSTA Technical Committee 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.

For the latest information on OSTA and UDF visit the OSTA web site at the following address:

<p align="center">http://www.osta.org</p>

OSTA Universal Disk Format Specification
Developer Registration Form

OSTA
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 UNIX/POSIX   O Windows NT   O Windows 95
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 DVD-ROM   O DVD-R   O DVD-RAM   O DVD-Video
O Other _____

Please indicate what value you plan to use in the *Implementation Identifier* field of the *Entity Identifier* descriptor to identify your implementation:

_____

NOTE: The identifier should be something that uniquely identifies your company as well as your product.

O Please add my email address to the OSTA Technical Committee email reflector.

O Please send an OSTA Membership kit.

FAX Completed form to OSTA at 1-805-962-1541, or mail to:
OSTA, 311 E. Carrillo Street, Santa Barbara, CA 93101

111

U

Unallocated Space Descriptor, 7, 20
Unicode, 8, 9, 67, 68, 80
UniqueID, 21, 38, 39, 45, 49, 52, 61, 62, 63, 76, 77, 108
UNIX, 46, 48, 63, 72, 73

V

VAT, 5, 25, 26, 27, 28, 51, 100, 101, 102

Virtual Allocation Table, 5, 26, 27, 28
virtual partition, 25, 27, 101
Virtual Partition Map, 25

W

Windows, 46, 47, 57, 69
Windows 95, 46, 47, 72, 78, 110
Windows NT, 46, 47, 57, 72, 78, 79, 90, 110
WORM, 6, 20, 32

OSTA Universal Disk Format Specification
          Revision History
=======================================
1.50 02/04/97 Added support for CD-R Packet Writing
                Added support for CD-RW Packet Writing
                Added support for Windows 95
                Added limited support for Windows NT
                Incorporates Document Change Notices (DCN)
                DCN 2-025 through 2-033
1.02 10/30/96 Incorporates Document Change Notices (DCN)
                DCN2-001 through 2-024.
1.01 11/03/95 Added DVD Apendix and made a few
                minor editoral changes.
1.00 10/24/95 Original Release

```
/**************************************************************************
 * 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);
}
```

```
/***********************************************************************
 * OSTA UDF compliant file name translation routine for DOS.
 * 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>

#define DOS_NAME_LEN     8
#define DOS_EXT_LEN      3
#define ILLEGAL_CHAR_MARK 0x005F
#define CRC_MARK          0x0023
#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 short unicode_t;
typedef unsigned char byte;

/*** PROTOTYPES ***/
unsigned short cksum(register unsigned char *s, register int n);
int IsIllegal(unicode_t current);

/* Define functions or macros to both determine if a character is printable
 * and compute the uppercase version of a character under your implementation.
 */
int UnicodeIsPrint(unicode_t);
unicode_t UnicodeToUpper(unicode_t);

/***********************************************************************
 * Translate udfName to dosName using OSTA compliant.
 * dosName must be a unicode string with min length of 12.
 *
 * RETURN VALUE
 *    Number of unicode characters in dosName.
 */
int UDFDOSName(
unicode_t *dosName,   /* (Output) DOS compatible name.   */
unicode_t *udfName,   /* (Input)  Name from UDF volume.  */
int        udfLen,    /* (Input)  Length of UDF Name.    */
byte      *fidName,   /* (Input)  Bytes as read from media.              */
int        fidNameLen)/* (Input)  Number of bytes in fidName.            */
{
    int index, dosIndex = 0, extIndex = 0, lastPeriodIndex;
    int needsCRC = FALSE, hasExt = FALSE, writingExt = FALSE;
    unsigned short valueCRC;
    unicode_t ext[DOS_EXT_LEN], current;

    /*Used to convert hex digits. Used ASCII for readability. */
    const char hexChar[] = "0123456789ABCDEF";

    for (index = 0 ; index < udfLen ; index++)
    {
        current = udfName[index];
        current = UnicodeToUpper(current);
```

```c
        if (current == PERIOD)
        {
            if (dosIndex==0 || hasExt)
            {
                /* Ignore leading periods or any other than used for extension. */
                needsCRC = TRUE;
            }
            else
            {
                /* First, find last character which is NOT a period or space. */
                lastPeriodIndex = udfLen - 1;
                while (lastPeriodIndex >= 0 && (udfName[lastPeriodIndex] == PERIOD
                                            || udfName[lastPeriodIndex] == SPACE))
                {
                    lastPeriodIndex--;
                }

                /* Now search for last remaining period. */
                while (lastPeriodIndex >= 0 && udfName[lastPeriodIndex] != PERIOD)
                {
                    lastPeriodIndex--;
                }

                /* See if the period we found was the last or not. */
                if (lastPeriodIndex != index)
                {
                    needsCRC = TRUE; /* If not, name needs translation. */
                }

                /* As long as the period was not trailing,
                 * the file name has an extension.
                 */
                if (lastPeriodIndex >= 0)
                {
                    hasExt = TRUE;
                }
            }
        }
        else
        {

            if ((!hasExt && dosIndex == DOS_NAME_LEN) || extIndex == DOS_EXT_LEN)
            {
                /* File name or extension is too long for DOS. */
                needsCRC = TRUE;
            }
            else
            {
                if (current == SPACE)    /* Ignore spaces. */
                {
                    needsCRC = TRUE;
                }
                else
                {
                    /* Look for illegal or unprintable characters. */
                    if (IsIllegal(current) || !UnicodeIsPrint(current))
                    {
                        needsCRC = TRUE;
                        current = ILLEGAL_CHAR_MARK;
                        /* Skip Illegal characters(even spaces), but not periods. */
                        while(index+1 < udfLen
```

```
                            && (IsIllegal(udfName[index+1])
                                    || !UnicodeIsPrint(udfName[index+1]))
                            && udfName[index+1] != PERIOD)
                    {
                        index++;
                    }
                }

                /* Add current char to either file name or ext. */
                if (writingExt)
                {
                    ext[extIndex++] = current;
                }
                else
                {
                    dosName[dosIndex++] = current;
                }
            }
        }
    }
    /* See if we are done with file name, either because we reached
     * the end of the file name length, or the final period.
     */
    if (!writingExt && hasExt && (dosIndex == DOS_NAME_LEN ||
                    index == lastPeriodIndex))
    {
        /* If so, and the name has an extension, start reading it. */
        writingExt = TRUE;
        /* Extension starts after last period. */
        index = lastPeriodIndex;
    }
}

/*Now handle CRC if needed. */
if (needsCRC)
{
    /* Add CRC to end of file name or at position 4. */
    if (dosIndex >4)
    {
        dosIndex = 4;
    }

    dosName[dosIndex++] = CRC_MARK;
    valueCRC = cksum(fidName, fidNameLen);

    /* Convert lower 12-bits of CRC to hex characters. */
    dosName[dosIndex++] = hexChar[(valueCRC & 0x0f00) >> 8];
    dosName[dosIndex++] = hexChar[(valueCRC & 0x00f0) >> 4];
    dosName[dosIndex++] = hexChar[(valueCRC & 0x000f)];
}

/* Add extension, if any. */
if (extIndex != 0)
{
    dosName[dosIndex++] = PERIOD;
    for (index = 0; index < extIndex; index++)
    {
        dosName[dosIndex++] = ext[index];
    }
}

return(dosIndex);
```

```
}

/**************************************************************************
 * Decides if a Unicode character matches one of a list of ASCII characters.
 * Used by DOS 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);
}

/**************************************************************************
 * Decides whether character passed is an illegal character for a
 * DOS file name.
 *
 * RETURN VALUE
 *
 *     Non-zero if file character is illegal.
 */
int IsIllegal(
unicode_t ch)  /* (Input) character to test. */
{
    /* Genuine illegal char's for DOS. */
    if (ch < 0x20 || UnicodeInString("\\/:*?\"<>|", ch))
    {
        return(1);
    }
    else
    {
        return(0);
    }
}
```

```
/**********************************************************************
 * 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 cksum(unsigned char *s, 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.  */
byte *fidName,      /* (Input)  Bytes as read from media. */
int fidNameLen)     /* (Input)  Number of bytes in fidName. */
{
    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) || !isprint(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])
                                || !isprint(udfName[extIndex + index + 2])))
                    {
                        index++;
                    }
                }
                ext[localExtIndex++] = current;
            }

            /* Truncate filename to leave room for extension and CRC. */
            maxFilenameLen = ((MAXLEN - 4) - 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 = cksum(fidName, fidNameLen);
        /* 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
}
```

UDF Specification v1.02 - A specification
describing the Universal Disk Format
developed by the Optical Storage Technology
Association (OSTA). This specification is
for developers who plan to implement UDF
which is based upon the ISO 13346 standard.
UDF is a file system format standard that
enables file interchange among different
operating systems.