-OSTA-®
Optical Storage
Technology Association

Universal Disk Format™ Specification

Revision 1.01

November 3, 1995

Revision History:

1.00	October 24, 1995	Original Release
1.01	November 3, 1995	DVD appendix added

Optical Storage Technology Association
311 East Carrillo Street
Santa Barbara, CA 93101
(805) 963-3853 Voice
(805) 962-1541 Fax
osta@aol.com

This document along with the sample source code is available in electronic format from OSTA.

Important Notices

CONTENTS

1. INTRODUCTION 1

1.1 Document Layout 2

1.2 Compliance..3

2. BASIC RESTRICTIONS & REQUIREMENTS 4

2.1 Part 1 - General 6
2.1.1 Character Sets 6
2.1.2 OSTA CS0 Charspec 7
2.1.3 Timestamp 7
2.1.4 Entity Identifier..8

2.2 Part 3 - Volume Structure...13
2.2.1 Descriptor Tag 13
2.2.2 Primary Volume Descriptor 13
2.2.3 Anchor Volume Descriptor Pointer...15
2.2.4 Logical Volume Descriptor...16
2.2.5 Unallocated Space Descriptor 18
2.2.6 Logical Volume Integrity Descriptor 18
2.2.7 Implemention Use Volume Descriptor..20

2.3 Part 4 - File System...22
2.3.1 Descriptor Tag 22
2.3.2 File Set Descriptor 22
2.3.3 Partition Header Descriptor...25
2.3.4 File Identifier Descriptor...26
2.3.5 ICB Tag 27
2.3.6 File Entry 29
2.3.7 Unallocated Space Entry...30
2.3.8 Space Bitmap Descriptor...31
2.3.9 Partition Integrity Entry 31
2.3.10 Allocation Descriptors 31
2.3.11 Allocation Extent Descriptor..32
2.3.12 Pathname...33

2.4 Part 5 - Record Structure..33

3. SYSTEM DEPENDENT REQUIREMENTS 34

3.1 Part 1 - General 34
3.1.1 Timestamp 34

3.2 Part 3 - Volume Structure...35
3.2.1 Logical Volume Header Descriptor...35

3.3 Part 4 - File System 36
3.3.1 File Identifier Descriptor 36
3.3.2 ICB Tag 37
3.3.3 File Entry 39
3.3.4 Extended Attributes 44

4. USER INTERFACE REQUIREMENTS 54

4.1 Part 3 - Volume Structure 54

4.2 Part 4 - File System 54
4.2.1 ICB Tag 54
4.2.2 File Identifier Descriptor 55

5. INFORMATIVE 62

5.1 Descriptor Lengths 62

5.2 Using Implementation Use Areas 62
5.2.1 Entity Identifiers 62
5.2.2 Orphan Space 62

5.3 Boot Descriptor 63

5.4 Technical Contacts 63

6. APPENDICES 64

6.1 UDF Entity Identifier Definitions 64

6.2 UDF Entity Identifier Values 64

6.3 Operating System Identifiers 65

6.4 OSTA Compressed Unicode Algorithm 66

6.5 CRC Calculation 69

6.6 Algorithm for Strategy Type 4096 72

6.7 Identifier Translation Algorithms 73
6.7.1 DOS Algorithm 73
6.7.2 OS/2 , Macintosh and UNIX Algorithm 78

6.8 Extended Attribute Checksum Algorithm 83

6.9 Requirements for Digital Video Disc (DVD) 84
6.9.1 Constraints imposed by UDF for DVD 84
6.9.2 How to read a UDF disc 86

6.10 Developer Registration Form 89

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 6.10.

1.1 Document Layout

This document presents information on the treatment of structures defined under standard ISO/IEC 13346. The following areas are covered
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.

The standard ISO/IEC 13346 is commonly referred to as the NSR standard where NSR stands for “Non-Sequential Recording”. In this document we sometimes use the term NSR to refer to ISO/IEC 13346.

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, 3 and 4 of ISO/IEC 13346. Compliance to part 2 of ISO/IEC 13346 is optional. 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.

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 Rewritable and Overwritable media only, or WORM media only, or both. All implementations should be able to support Read-Only media.
- *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 support the extended attributes for the operating system they are currently running under. 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.

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

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 NSR structures. This area shall not be referenced by the Unallocated Space Descriptor or any other NSR descriptor. This is intended for the free use of 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
Primary Volume Descriptor	There shall be exactly one prevailing Primary Volume Descriptor recorded per volume.
Anchor Volume Descriptor Pointer	Shall only be recorded at 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 Partition Maps field shall contain only Type 1 Partition Maps.
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 Extents	The length of any single *Allocation Extent* 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.

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 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;
}
```

2.1.3.1 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 the 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.

2.1.4 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.

2.1.4.1 Uint8 Flags

👓 Self explanatory.

✍ Shall be set to ZERO.

2.1.4.2 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 current implementation shall set the *Identifier* field to a value that uniquely identifies the current implementation.

The following table summarizes the *Entity Identifier* fields defined in the NSR standard and shows the values they shall be set to.

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

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.

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*.

2.1.4.3 IdentifierSuffix

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

In regards 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 (= #0100)
2	1	Domain Flags	Uint8
3	5	Reserved	bytes (= #00)

The *UDFRevision* field shall contain **#0100** to indicate revision **1.00** 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 setable 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 This flag may be set/reset by the user. The *HardWriteProtect* flag is an implementation setable 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 permanently write protected. 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.

For 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 (= #0100)
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 (possibly non-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 the value in the prevailing Primary Volume Descriptor + 1 be used.

2.2.1.2 Uint16 DescriptorCRCLength

CRCs shall be supported and calculated for each descriptor. The value of this field shall be set to the 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_a d       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 shall be set to a unique value. The remainder of the field may be set to any allowed value.
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 the section on *Entity Identifier.*

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 only be recorded at 2 of the following 3 locations on the media :

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

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 the section on *Entity Identifier.*

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.

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 anytime the contents of the associated Logical Volume is modified. Through the contents of the *Logical Volume Integrity Descriptor* an implementation can easily answer the following useful questions:

1) Are the contents of the Logical Volume in a consistent state?

2) When was the last date and time that anything within the Logical Volume was modified?

3) What is the total Logical Volume free space in logical blocks?

4) What is the total size of the Logical Volume in logical blocks?

5) What is the next available UniqueID for use within the Logical Volume?

6) Has some *other* implementation modified the contents of the logical volume since the last time that the *original* implementation which created the logical volume accessed it.

2.2.6.1 byte LogicalVolumeContentsUse

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: Only with a closed *Logical Volume Integrity Descriptor* are you guaranteed to have a correct FreeSpaceTable.

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	??	Implementation Use	byte

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

Number of Files - The current number of files in the 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.

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.

2.3 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 non-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 the 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             FileSetIdent ifer[32];
        dstring             CopyrightFileIdentifier[32];
        dstring             AbstractFileIdentifier[32];
        struct long_ad      RootDirectoryICB;
        struct EntityID     DomainIdentifier ;
        struct long_ad      NextExtent;
        byte                Reserved[48];
}
```

On rewritable/overwritable media, only one *FileSet* descriptor shall be recorded. On WORM media, multiple *FileSet* descriptors 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 0's since PartitionIntegrityEntrys are not used.

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 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 shall indicate (ZERO) that the file is relocatable.

✍ Shall be set to ZERO.

Bit 9 (*Contiguous*):

👓 For OSTA UDF compliant media this bit shall indicate (ZERO) that the file may be non-contiguous.

✍ Shall 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 shall 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 also have its own *UniqueID*.

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.

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.

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.

2.3.12 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 1.

✍ Shall be set to 1.

2.4 Part 5 - Record Structure

With respect to this document *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[??];
}
```

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, 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 5.4.2.1., 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

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**	**Mac OS**	**UNIX**
Read	file	The file may be read	E	E	E	E
Read	directory	The directory may be read	E	E	E	E
Write	file	The file’s contents may be modified	E	E	E	E
Write	directory	Files or subdirectories may be created, deleted or renamed	E	E	E	E
Execute	file	The file by be executed.	I	I	I	E
Execute	directory	The directory may be searched for a specific file or subdirectory.	E	E	E	E
Attribute	file	The file’s permissions may be changed.	E	E	E	E
Attribute	directory	The directory’s permissions may be changed.	E	E	E	E
Delete	file	The file may be deleted.	E	E	E	E
Delete	directory	The directory may be deleted.	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	Mac OS	UNIX
Read	file	The file may be read	1	1	1	U
Read	directory	The directory may be read, only if the directory is also marked as *Execute*.	1	1	1	U
Write	file	The file's contents may be modified	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
Execute	file	The file by be executed.	0	0	0	U
Execute	directory	The directory may be searched for a specific file or subdirectory.	1	1	1	U
Attribute	file	The file's permissions may be changed.	1	1	1	Note 1
Attribute	directory	The directory's permissions may be changed.	1	1	1	Note 1
Delete	file	The file may be deleted.	Note 2	Note 2	Note 2	Note 2
Delete	directory	The directory may be deleted.	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 be the end of the extended attribute s space.

3.3.4.2 Alternate Permissions

```
struct AlternatePermissionsExtendedAttribute {      /*    ISO    13346
4/14.10.4 */
        Uint32          AttributeTyp e;
        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;
        Uint32          MajorDeviceIdentification ;
        Uint32          MinorDeviceIdentification;
        byte            ImplementationUse[2];
}
```

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.

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 attribute s.

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 are currently running under. 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

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.

3.3.4.5.2 MS-DOS

👓 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 attribute s 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.

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
1	2	HeaderChecksum	Uint16
3	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.

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 the MS-DOS and OS/2 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 prior to the *last "."* (#002E) shall be considered as constituting the *file name*, characters appearing subsequent to the *last "."* (#002E) shall be considered as constituting the *file extension*.

> **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 prior to the first *"."* (#002E) character shall be considered as constituting the *file name* and all characters appearing

subsequent to the last *"."* (#002E) character shall be considered as constituting the *file extension*. All embedded *"."* (#002E) characters within the *file name* shall be removed.

7. Long Extension: In the event that the number of characters constituting the *file extension* at this step in the process is greater than 3, the *file extension* shall be regarded as having been composed of the first 3 characters amongst the characters constituting the *file extension* at this step in the process.
8. Long Filename: In the event that the number of characters constituting the file name at this step in the process is greater than 8, the *file name* shall be truncated to 4 characters.
9. FileIdentifier CRC: Since through the above process character information from the original *FileIdentifier* is lost the chance of creating a duplicate *FileIdentifier* in the same directory increases. To greatly reduce the chance of having a duplicate *FileIdentifier* the file name shall be modified to contain a CRC of the original *FileIdentifier*. The *file name* shall be composed of the first 4 characters constituting the *file name* at this step in the process, followed by the separator “#” (#0023); followed by a 3 digit CS0 Hex representation of the least significant 12 bits of the 16-bit CRC of the original CS0 *FileIdentifier*.
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*. The new *FileIdentifier* shall be composed of up to the first (250 - (length of *file extension* +1) - 5) characters constituting the *FileIdentifier* at this step in the process, followed by the separator "#" (#0023); followed by a 3 digit CS0 Hex representation of the least significant 12 bits of the 16-bit CRC of the original CS0 *FileIdentifier*, followed by the first 5 characters of the file extension including the "."(#002E).

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 *27* 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*. The new *FileIdentifier* shall be

composed of up to the first (27 - (length of *file extension* +1) -5) characters constituting the *FileIdentifier* at this step in the process, followed by the separator “#” (#0023); followed by a 3 digit CS0 Hex representation of the least significant 12 bits of the 16-bit CRC of the original CS0 *FileIdentifier*, followed by the first 5 characters of the file extension including the “.”(#002E).

4.2.2.1.4 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. <u>*FileIdentifier* Lookup:</u> Upon request for a *"lookUp"* of a *FileIdentifier*, a case-sensitive comparison shall be performed.
2. <u>Validate *FileIdentifer*:</u> If the *FileIdentifier* is a valid UNIX file identifier for the current system environment then do not apply the following steps.
3. <u>Invalid Characters:</u> 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. <u>Long FileIdentifier</u> - 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-4* characters of the *FileIdentifier* at this step in the process *.*
5. <u>FileIdentifier CRC:</u> 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 checksum of the original *FileIdentifier*. The new *FileIdentifier* shall be composed of upto the first ((*MAXNameLength-4*) - (length of *file extension* +1) -5) characters constituting the *FileIdentifier* at this step in the process, followed by the separator “#” (#0023); followed by a 3 digit CS0 Hex representation of the least significant 12 bits of the 16-bit CRC of the original CS0 *FileIdentifier*, followed by the first 5 characters of the file extension including the “.”(#002E).

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 **osta @aol.com** . 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

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 extended attribute space.
"*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.

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 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

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
5-255	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
2	0	OS/2
3	0	Macintosh OS
4	0	UNIX - Generic
4	1	UNIX - IBM AIX
4	2	UNIX - SUN Solaris
4	3	UNIX - HP/UX
4	4	UNIX - Silicon Graphics Irix

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 Windows 95 , Windows NT and NetWare are not 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;
         }
         if (byteIndex < numberOfBytes)
         {
            /*Then the next byte to the low bits. */
            unicode[unicodeIndex] |= UDFCompressed[byteIndex++];
         }
         unicodeIndex++;
      }
      returnValue = unicodeIndex;
   }
   return(returnValue);
}


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

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

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

   return(byteIndex);
}
```

6.5 CRC Calculation

The following C program may be used to calculate the CRC -CCITT checksum used in the TAG descriptors of ISO/IEC 13346.

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

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

```
#include    <stdio.h>

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

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

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

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

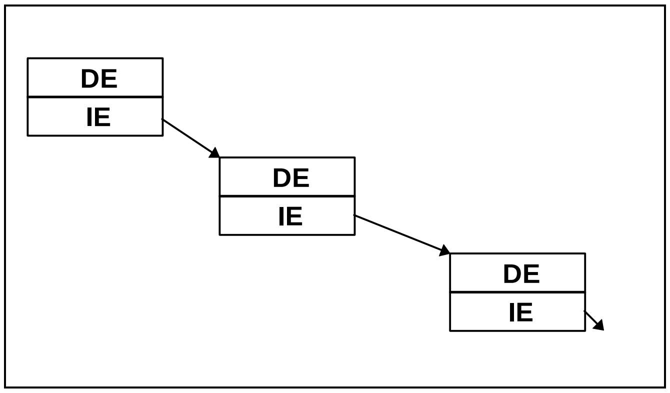
All the above CRC code was devised by Don P. Mitchell of AT&T Bell Laboratories and Ned W. Rhodes of Software Systems Group.
It has been published in "Design and Validation of Computer Protocols", Prentice Hall, Englewood Cliffs, NJ, 1991, Chapter 3, ISBN 0-13-539925-4.
Copyright is held by AT&T.

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 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);

      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);
   }
}
```

6.7.2 OS/2 , Macintosh and UNIX Algorithm

```
/***********************************************************************
 * OSTA UDF compliant file name translation routine for OS/2,
 * 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
 *
 * 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
   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
      /* 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
   /* For OS2, 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 - 4)
      {
         /*If no extension, make sure to leave room for CRC. */
         newIndex = MAXLEN - 4;
      }
      newName[newIndex++] = CRC_MARK; /* Add mark for CRC. */

      /*Calculate CRC from original filename from FileIdentifier. */
      valueCRC = cksum(fidName, fidNameLen);
      /* Convert lower 12-bits of CRC to hex characters. */
      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
/***********************************************************************
 * 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
   /* 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 Digital Video Disc (DVD)

This appendix defines the requirements and restrictions for UDF formatted media for dedicated Digital Video Disc (DVD) content players. DVD is an application of the 2nd generation (high capacity) CD-ROM, CD Recordable and CD Erasable media for the home consumer market.

All DVD discs shall be mastered to contain all required data as specified by ISO 13346 and UDF. This will allow playing of DVD in computer systems. Examples of such data include the time, date, permission bits, and a free space map (indicating no free space if ROM media). While DVD player implementations may ignore these fields, a UDF computer system implementation will not.

This appendix also shows the basics for extracting the necessary information needed by a DVD player from a UDF formatted disc. While the UDF format has many features and can be complicated, much of the information can be ignored in a read only, dedicated DVD player environment. Both entertainment-based (DVD) 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.

6.9.1 Constraints imposed by UDF for DVD

This section describes the restrictions and requirements for UDF formatted DVD discs. Due to limited computing resources within a DVD consumer player these restrictions and requirements were created so that a DVD consumer player would not have to support every feature of the UDF specification.

- The DVD files should be stored in a subdirectory directly under the root directory. This directory name should be standardized in the *DVD Application Specification*.

  The remainder of this document will use an example directory name of the five characters "DVD_2"

  NOTE: The *DVD Application Specification* is a document, which will be developed by the DVD manufacturers, that describes the names of the DVD files and directories which will be stored on the media, and additionally describes the contents of the DVD files

- A control or index file should be written in the aforementioned subdirectory. This file name should be standardized in the *DVD Application Specification*.

The remainder of this document will use an example file name of the seven characters "CONTROL"

- All other file names should be specified within the CONTROL file.

- All references to blocks of data shall be of the form of a filename and a relative byte offset into that file.

- A DVD player shall not be required to follow symbolic links to any files.

- File names shall consist of 8 bit Unicode characters. When the DVD is recorded with 16 bit Unicode characters, the DVD file names shall consist of the first 256 characters of Unicode. i.e. all but the lower 8 bits of a character are set to zero.
  It is recommended that DVD file names consist of the d-character set defined in ISO 646. This consists of the characters 'A'-'Z' (upper case), '0'-'9' (digits), '_' (underscore), and '.' (full stop). It is further recommended that not more than one '.' be used.
  Maximum compatibility will be ensured if the DVD file names consist of no more than eight characters, optionally followed by a '.', optionally followed by no more than three characters.

- File name comparisons shall be performed in a case sensitive manner. A simple byte for byte comparison may be used. If a matching file identifier is not found, a case insensitive comparison will then be used. This comparison can be performed by "upcasing" both strings before performing a byte by byte comparison.

- There shall be no identifiers that differ from identifiers specified in this appendix only by case. i.e. there shall not be a dvd_2 directory in the root. There shall not be any files in the DVD _2 directory that differ only by case. i.e. 'Movie' and 'movie' cannot coexist.

- Originating systems shall constrain individual files to be less than 2^31 (2G) in length. (maximum size 2^31 - 1). This constraint allows use of 32 bit signed pointers.
  Note: If the last sector of a file must contain all valid bytes, the maximum file size shall be 2^31 - blocksize).

- Other applications should define appropriate directory names, i.e. AUDIO.2, GAME.MODEL_X, BOOK, etc. in their respective standards.

All the above constraints apply only to the directory and files to 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 (DVD) and computer-based content on the same disc.

6.9.2 How to read a UDF disc

The following section describes the basic steps that a DVD   player would go through to read a UDF formatted DVD disc.

6.9.2.1 Find Anchor Point

The 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. The anchor point is the root pointer to everything on the disc.
A DVD player only needs to look at logical sector 256; the copy at 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 in locations 0-3 and 5 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.

6.9.2.2 Find the Logical Volume Descriptor

Read logical sectors:

MVDS_Location through MVDS_Location + (MVDS_Length - 1) / sectorSize

The logical sector size will depend on the implementation; it will not have an arbitrary size on a per disc basis. As currently specified, all logical sector sizes shall be 2048 bytes for DVD media. The Logical Volume Descriptor shall be the first logical sector that contains a descriptor with a tag identifier of 6.

FSD_Partition, FSD_Location, and FSD_Length are obtained from this structure.

6.9.2.3 Find the Partition Descriptors

Read logical sectors <pre>

MVDS_Location through MVDS_Location + (MVDS_Length - 1) / blocksize

The Partition Descriptors shall have a descriptor with a tag identifier of 5. The partition number and partition location shall be recorded for the first two partitions found. If not found, this disc is no good.

Partition_Location and Partition_Length are returned. Note that the location is absolute and the length is in logical blocks.

6.9.2.4 Read File Set Descriptor

All logical block numbers within a file set are relative. The base for computing absolute logical block numbers is the Partition_Location. Add Partition_Location to all relative values to get the absolute logical block number.

The File Set Descriptor is located at absolute addresses

Partition_Location + FSD_Location through
Partition_Location + FSD_Location + (FSD_Length - 1) / blocksize

The entire extent should be read. As each File Set Descriptor is found, it shall be made the current descriptor.

After the last File Set Descriptor in the extent is found, also search the extent starting at NextExt_Location and NextExt_Length in size. The NextExt_partition variable should be examined to see that it matches the current partition value. Repeat this process until the NextExt_Location is 0 or the NextExt_Location points to a blank area.

The repetitive read is to allow for updating of write once media.

RootDir_Partition, RootDir_Location and RootDir_Length shall be read from the File Set Descriptor.

6.9.2.5 Read Root Directory ICB

RootDir_Location and RootDir_Length define the location of an ICB extent. This ICB extent contains a File Entry and possibly an Indirect Entry to another ICB extent. The chain of indirect entries shall be followed to its end. The File Entry describes the data space and permissions of the root directory. The Indirect Entries allow for updating files on write-once, limited overwrite, and mixed ROM/RAM media.

A list of extents containing the Root Directory is returned.

6.9.2.6 Find Subdirectory Entry

Parse the data in the root directory extents to find the movie subdirectory.

Find the DVD _2 File Identifier Descriptor. The name shall be in 8, 16 or 32 bit compressed UDF format (essentially uncompressed). Verify that DVD_2 is a directory.

6.9.2.7 Find Subdirectory

The ICB found in the step above contains another File Entry. This File Entry shall be a directory, and its extents contain the list of files in the DVD _2 subdirectory. *The location of this File Entry should be retained* for finding arbitrary video files later.

6.9.2.8 Find the Control File Descriptor

The extents found in the step above contain sets of File Identifier Descriptors. In this pass, verify that the entry points to a file and is named CONTROL.

Further files can be found in the same manner as the CONTROL file when needed. Other file names shall be specified only in the CONTROL file. The names shall be pre-compressed in the CONTROL file which maximizes computer compatibility without requiring the player to understand any compression algorithms.

NOTE: Initially DVD discs should be mastered with both the UDF and ISO 9660 file systems. This *UDF Bridge* disc will allow playing DVD media in computers immediately 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. The DVD consumer players shall only support UDF and not ISO 9660.

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 6.10. For planned operating system check the *Other* box and write in DVD.

6.10 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 willing to interchange media with you to verify data interchange between your implementation and their implementation.
- 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.

OSTA
Optical Storage
Technology Association

OSTA Universal Disk Format Specification Developer Registration Form

Name: ____________________ ____________________

Company: ____________________ ____________________

Address: ____________________ ____________________

____________________ ____________________

____________________ ____________________

City: ____________________ ____________________

State/Province: ____________________ ____________________

Zip/Postal Code: ____________________ ____________________

Country: ____________________ ____________________

Phone: ____________________ FAX: ____________________

Email: ____________________ ____________________

Planned Operating Systems Support
Please indicate on which operating systems you plan to support ISO/IEC 13346:

DOS	OS/2	Macintosh
UNIX/POSIX	Windows NT	Windows 95

Other ____________________

____

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

____________________ ____________________

Miscellaneous

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

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

—A—

Allocation Descriptor, 5, 27, 31, 32
Allocation Extent Descriptor, 32
Anchor Volume Descriptor Pointer, 4, 15

—C—

Charspec, 7
Checksum, 47, 48, 49, 50, 52, 82
CRC, 13, 22, 31, 68, 70
CS0, 6, 7, 10, 14, 15, 16, 21, 24, 54, 56, 58, 59, 60

—D—

Descriptor Tag, 13, 22, 31
Domain, 1, 8, 9, 10, 11
DOS, 36, 37, 47, 54, 57
DVD, 2, 83, 84, 85, 86, 87

—E—

Entity Identifier, 4, 8, 9, 13, 14, 15, 16, 17, 19, 20, 23, 24, 25, 26, 29, 30, 31, 39, 46, 63
Extended Attributes, 3, 20, 43, 44, 46, 47, 48, 49, 50, 51, 52, 53, 63

—F—

File Entry, 5, 9, 29, 39, 45, 52, 63
File Identifier Descriptor, 9, 26, 36, 55
File Set Descriptor, 5, 9, 22, 24

—H—

HardWriteProtect, 11, 17, 23, 25

—I—

ICB, 5, 26, 27, 36, 37, 43, 54, 55
ICB Tag, 5, 27, 37, 54
ImplementationIdentifier, 14, 15, 16, 17, 20, 24, 29, 30, 31, 39, 46, 47, 48, 49, 51, 52
ISO/IEC 13346, 1

—L—

Logical Block Size, 4, 5, 17
Logical Sector Size, 4
Logical Volume Descriptor, 5, 9, 16, 18, 19
Logical Volume Header Descriptor, 19, 35
Logical Volume Integrity Descriptor, 9, 17, 18, 31

—M—

Macintosh, 30, 35, 36, 38, 43, 45, 49, 50, 51, 52, 59, 63

—N—

NetWare, 64

—O—

Orphan Space, 61
OS/2, 36, 37, 48, 49, 54, 55, 58, 63
Overwritable, 3, 4

—P—

Partition Header Descriptor, 25
Partition Integrity Entry, 5, 9, 31
Pathname, 33
Primary Volume Descriptor, 4, 9, 13

—R—

Read-Only, 3, 4
Records, 5, 33
Rewritable, 3, 4, 25, 32

—S—

SoftWriteProtect, 11, 17, 25
strategy, 5, 23, 27
SymbolicLink, 54

—T—

Timestamp, 4, 7, 18, 34

—U—

Unallocated Space Descriptor, 5, 18
Unicode, 6, 7, 55, 56, 65
UniqueID, 18, 29, 30, 35, 39, 43, 51, 52, 63
UNIX, 36, 38, 53, 59, 60

—W—

Windows, 36, 37, 47, 54, 57
Windows 95, 64
Windows NT, 64
WORM, 3, 4, 17, 23