INTERNATIONAL STANDARD

ISO/IEC 14496-18

First edition
2004-07-01

Information technology — Coding of audio-visual objects —

Part 18: Font compression and streaming

Technologies de l'information — Codage des objets audiovisuels —

Partie 18: Compression et transmission de polices de caractères

Reference number
ISO/IEC 14496-18:2004(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword iv

Introduction vi

1 **Scope** 1

2 **Normative references** 1

3 **Font Data Format** 1

4 **Font Compression Technology** 2
4.1 **Overview** 2
4.2 **Compressed Font Format Specification** 2

5 **Font Data Stream** 12
5.1 **Structure of the Font Data Stream** 12
5.2 **Access Unit Definition** 12
5.3 **Time Base for Font Data Streams** 14
5.4 **Font Data Decoder Configuration** 14
5.5 **Accessing the Font Data** 15

6 **Text Profiles and Levels** 15
6.1 **Simple Text Profile and Levels** 15
6.2 **Advanced Simple Text Profile and Levels** 16
6.3 **Main Text Profile and Levels** 17

Annex A (informative) **Patent Statements** 18

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 14496-18 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 14496 consists of the following parts, under the general title *Information technology — Coding of audio-visual objects*:

— *Part 1: Systems*

— *Part 2: Visual*

— *Part 3: Audio*

— *Part 4: Conformance testing*

— *Part 5: Reference software*

— *Part 6: Delivery Multimedia Integration Framework (DMIF)*

— *Part 7: Optimized reference software for coding of audio-visual objects*

— *Part 8: Carriage of ISO/IEC 14496 contents over IP networks*

— *Part 9: Reference hardware description*

— *Part 10: Advanced Video Coding*

— *Part 11: Scene description and application engine*

— *Part 12: ISO base media file format*

— *Part 13: Intellectual Property Management and Protection (IPMP) extensions*

— *Part 14: MP4 file format*

— *Part 15: Advanced Video Coding (AVC) file format*

— *Part 16: Animation Framework eXtension (AFX)*

— *Part 17: Streaming text format*

— *Part 18: Font compression and streaming*

— *Part 19: Synthesized texture stream*

Introduction

ISO/IEC 14496 specifies a system for the communication of interactive audio-visual scenes. The specification includes the following elements:

1. the coded representation of natural or synthetic, two-dimensional (2D) or three-dimensional (3D) objects that can be manifested audibly and/or visually (audio-visual objects) (specified in part 1,2 and 3 of ISO/IEC 14496);

2. the coded representation of the spatio-temporal positioning of audio-visual objects as well as their behaviour in response to interaction (scene description, specified in part 11 of ISO/IEC 14496);

3. the coded representation of information related to the management of data streams (synchronization, identification, description and association of stream content, specified in part 11 of ISO/IEC 14496);

4. a generic interface to the data stream delivery layer functionality (specified in part 6 of ISO/IEC 14496);

5. an application engine for programmatic control of the player: format, delivery of downloadable Java byte code as well as its execution lifecycle and behaviour through APIs (specified in part 11 of ISO/IEC 14496); and

6. a file format to contain the media information of an ISO/IEC 14496 presentation in a flexible, extensible format to facilitate interchange, management, editing, and presentation of the media.

The information representation, specified in ISO/IEC 14496-1 and in ISO/IEC 14496-11, describes the means to create an interactive audio-visual scene in terms of coded audio-visual information and associated scene description information. The encoded content is presented to a terminal as the collection of elementary streams. Elementary streams contain the coded representation of either audio or visual data or scene description information or user interaction data. Elementary streams may as well themselves convey information to identify streams, to describe logical dependencies between streams, or to describe information related to the content of the streams. Each elementary stream contains only one type of data.

Elementary streams are decoded using their respective stream-specific decoders. The audio-visual objects are composed according to the scene description information and presented by the terminal's presentation device(s). All these processes are synchronized according to the systems decoder model (SDM) using the synchronization information provided at the synchronization layer.

The scene description stream identifies different types of objects, such as audio, visual, 2D and 3D graphics, etc. that define a scene composition of the content. Among these objects, the essential part of almost any multimedia presentation is text objects that are created utilizing specific custom fonts. Font selection determines the appearance of a text in multimedia content and it's the most critical factor that assures text legibility and readability. It also plays critical role in the overall scene composition since the metric properties of a font are used for textual parts of multimedia content layout. Many thousands of fonts are available today for use in content creation and in order to assure correct appearance and layout of a content the font data have to be included (embedded) with the text objects as part of the multimedia presentation.

Font data compression and streaming technology presented in this document provide efficient mechanism to embed font data in MPEG-4 encoded presentations.

Information technology — Coding of audio-visual objects —

Part 18:
Font compression and streaming

1 Scope

This part of ISO/IEC 14496 specifies functionalities for the communication of font data as part of the MPEG-4 encoded audio-visual presentation. More specifically, it defines:

1. Font format representation that is utilized for font data encoding (OpenType);
2. Font compression technology for TrueType and OpenType fonts with TrueType outlines; and
3. The coded representation of information in font data streams.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14496-1, *Information technology — Coding of audio-visual objects — Part 1: Systems*

ISO/IEC 14496-11, *Information technology — Coding of audio-visual objects — Part 11: Scene description and application engine*

The OpenType Specification - available on the Microsoft Typography website at <http://www.microsoft.com/typography/otspec/default.htm> or the Adobe Solutions Network website at <http://partners.adobe.com/asn/tech/type/opentype/index.jsp>

3 Font Data Format

In order to guarantee the original appearance of the content, to preserve corporate branding and identity in streaming multimedia presentations and to provide support for all languages, MPEG-4 supports text rendering utilizing rich formatting capabilities and custom fonts.

MPEG-4 adopts OpenType®[1)], version 1.4, as its font data format for the purposes of uniform font data transmission and predictable text rendering. OpenType has emerged as the font solution for high-quality text processing, multimedia applications and cross platform Internet document portability. OpenType is a full-featured font format that enables the highest quality of text rendering on low-resolution displays, advanced typographic features and international character support. It is fully compatible with the existing and widely adopted TrueType™[2)] fonts.

MPEG-4 requires fonts to contain a Unicode character map ('cmap') table.

1) OpenType is a registered trademark of Microsoft Corporation.

2) TrueType is a trademark of Apple Computer Incorporated.

4 Font Compression Technology

4.1 Overview

Transmission of the font data through bandwidth-limited channels requires efficient compression techniques be applied that allow reducing font data size without sacrificing quality of text rendering and advanced typographic features supported by the font. MPEG-4 adopts MicroType® Express[3)] 2.0 font compression technology, which allows lossless compression of OpenType fonts with TrueType outlines and provides the capability to render the text without decompressing the whole font, which significantly reduces memory requirements. The specification of this compression mechanism is presented below.

4.2 Compressed Font Format Specification

4.2.1 Introduction

This part of the document describes the compressed font format that is designed for compact binary representation of lossless-compressed TrueType and OpenType with TrueType outlines fonts (TTF) and font collections (TTC). This font compression technology for OpenType TrueType fonts is designed to minimize memory usage for font data storage in resource-constrained environments, where both ROM and RAM memory savings are critical, and build relatively small font files without sacrificing the quality, features and capabilities of TrueType and OpenType fonts.

The compressed font format provides random access capabilities into the font data and allows font rasterizer to render glyphs directly from the compressed font tables without decompressing the whole font while maintaining the highest character quality. The selection and number of compressed tables in a font file may differ from one font to another. Any number of tables can be compressed in the same font (from only one to all), which allows application developers to optimize “memory savings – performance of font rendering engine” trade-off.

Compression algorithms, applied to font tables, take advantage of advanced knowledge of OpenType font tables structure and meaning of data (text, TrueType instructions, glyph data, etc.). Separation of these objects into groups with distinctly different statistical distributions allows employing a number of compression techniques, which results in producing most compact lossless-compressed data. These techniques are especially effective when compressing of the glyph data (‘glyf’ table), which is typically the biggest table in the TTF file. The details of the glyph data compression are described in subclause 4.2.4 of this specification.

4.2.2 Data Types

The compressed font format follows the same SFNT structure (used in both TrueType and OpenType) that provides a “wrapper” for a collection of tables in a general and extensible manner. Rasterizers use a combination of tables contained in the font to render glyph outlines. In order to simplify the process of traversing the tables in a font, the beginning of each table is aligned on the 4-byte boundary and each table data is padded with zeros.

Most tables have version numbers, and the version number for the entire font is contained in the Table Directory. The compressed font format specification uses the same data types that are used in the OpenType specification. For all details on data types and version number format please see the OpenType font file specification.

3) MicroType is a registered trademark of Agfa Monotype Corporation.

This information is given for the convenience of users of this International Standard and does not constitute an endorsement by ISO/IEC of the product named.

4.2.3 Compressed Font Format

4.2.3.1 The Compressed Font Header

The compressed font file format is the same as an OpenType font file, which always begins with the Offset Table followed by the Table Directory entries. If the font file is TTC file, the offset to each Table Directory is defined in TTC Header. If the font file contains only one font, the Offset Table will begin at byte 0 of the file. For details about an OpenType font Table directory and TTC header structure please see “Organization of an OpenType Font” in the OpenType font file specification.

4.2.3.2 OpenType Tables

The OpenType font file specification defines the set of required tables to be used in an OpenType font with TrueType outlines. When this font compression mechanism is used with OpenType fonts with TrueType outlines, the ACT3 additional table described in the following section is required. The tag describing this table is ‘**act3**’.

Other tables may be defined but are not required. OpenType fonts may also contain bitmaps of glyphs, in addition to outlines, as well as other optional tables that support vertical layout and advanced typographic features. For a complete list of OpenType font tables, their definitions and data structures see The OpenType font file specification.

4.2.3.3 ACT3 Table Structure

The ‘ACT3’ table defines the mapping of all tables, present in the font, to the collection of compressed and uncompressed blocks of data in a font file. Since the herein compressed font file format allows a mixture of both compressed and uncompressed data blocks to be present in the file, the mapping of the data blocks relative to the original OpenType TrueType table structure will be as follows:

- all consecutive uncompressed tables of the original font file will be referenced as a contiguous block of uncompressed data by a single entry in the ACT3 table;
- each compressed table of the original font file will be referenced as a block of compressed data by a single entry in the ACT3 table.

  NOTE: The ‘act3’ table is required to be present in the compressed font OpenType Table Directory. If the font file is TrueType collection, then the table will only be part of the first font Table Directory and will contain mapping information for all fonts in TrueType Collection font file.

Table 1 — ‘act3’ table syntax

Data Type	Syntax	Description and Comments
	/* ACT3 Table Header*/	
USHORT	TableLength	Length of the ‘act3’ table padded to 4-byte boundary
ULONG	CompFontSize	Size of the compressed font file
ULONG	UncompFontSize	Original size of the uncompressed font file after CTF compression was applied to glyph data.
ULONG	GlyfStart	Offset to the start of compressed ‘glyf’ table in uncompressed font file
ULONG	GlyfEnd	Offset to the position immediately following the end of compressed ‘glyf’ table in uncompressed font file
USHORT	NumEntries	Number of different compressed/uncompressed data blocks in the font file
	/* ACT3 Table Entries*/	
	for (i=0; i<numEntries; i++)	
	{	

ULONG	BlockStart	Offset to the start of uncompressed data block in original font file
ULONG	BlockLength	Length of uncompressed data block in original font file
ULONG	CompBlockStart	Offset to the start of data block in the compressed font file
USHORT	IsCompressed	Flag indicating whether block is compressed (non-zero if block is compressed, **INTERVAL** = isCompressed)
}		

4.2.3.4 Structure of Compressed Block

Each compressed table in the original OpenType TrueType font or font collection file is represented by the single block of data in the ACT3 table. The data is losslessly compressed using canonical Huffman codes. When the table is compressed, an additional set of data is generated in order to enable the decoder perform random access into compressed data stream and selectively decompress only those chunks of data that are requested by the font rendering engine.

Random access in the compressed dataset is achieved by providing additional information on the corresponding positions of the start of Huffman codeword in the compressed data stream (bitmarks) and the start of a token in the uncompressed data. This relationship is established through predefined intervals between reference points in the original data.

The size of the interval between reference points affects both the compression efficiency of the encoder (a smaller interval would require more data be added to the output compressed data stream) and font rendering performance (a bigger interval would require font engine to decompress larger dataset in order to read the requested data). Therefore, a careful consideration should be given to selection of the interval between random access entry points. The analysis of the TrueType features, their sizes and empirical rendering performance results for Latin fonts showed that the optimal placement of reference points is achieved at the **INTERVAL** = 128 bytes.

The block of compressed data has **RAC3** tag and consists of the following data segments:

- The dictionary of tokens which were defined by the encoder;
- Definition of the canonical Huffman codes;
- Locations of the random access pointers to the compressed bitstream and original data;
- Compressed data bitstream.

Table 2 — RAC3 block format

Data Type	Syntax	Description and Comments
	/* RAC3 Block Header */	
TAG	RACtag	Compressed data ID string: 'R' 'A' 'C' 3 (0x52414303)
ULONG	compDataSize	Size of the compressed data
ULONG	uncompDataSize	Original size of the uncompressed block of data
	/* RAC3 Dictionary */	
ULONG	numTokens	Number of tokens in the dictionary.
BYTE	tokenLength[numTokens]	Array of length for each token in the dictionary (in bytes).
BYTE	offsetLength	Length of the tokenOffset from the beginning of the tokenString array to the start of the token (in bytes)
	if (offsetLength ==0)	Tokens are non-overlapping, each tokenOffset into the tokenString can be inferred from the array of tokenLength[].
	{	

ULONG	tokenStringSize	Length of the tokenString in bytes.
BYTE	tokenString[tokenStringSize]	Dictionary of the tokens.
	} else	We have overlapping tokens, offsets into the tokenString are explicitly specified in the tokenOffset[].
	{	
BYTE	tokenOffset [(numTokens*offsetLength)]	Array of offsets for each token in the tokenString. Each element of the array is offsetLength bytes long.
ULONG	tokenStringSize	Length of the tokenString in bytes.
BYTE	tokenString[tokenStringSize]	Dictionary of the tokens.
	}	
	/* Canonical Huffman Codes */	
BYTE	numGroups	Number of Huffman code length groups (can be from 1 to 32)
BYTE	codeLength[numGroups]	Array of code length in each Huffman group.
ULONG	firstGroupCode[numGroups]	Array of the first code in each Huffman group.
ULONG	lastGroupCode[numGroups]	Array of the last code in each Huffman group.
	/* Random Access Pointers */	
ULONG	numIntervals	Number of **INTERVAL** byte intervals in the uncompressed source data
ULONG	bitMarks[numIntervals]	Bit offset to the start of the nearest Huffman codeword from the beginning of compressed data block corresponding to n^{th} interval.
BYTE	deltaBytes[numIntervals]	Offset to the start of the nearest Huffman token from the beginning of uncompressed source data = **INTERVAL** * n – deltaBytes[n]
ULONG	BitStringOffset	Bit offset to the start of the compressed bitString[] from the beginning of the compressed data block.
BYTE	BitString[compDataSize-bitStringOffset/8]	Compressed data bitstream.

4.2.4 Glyph Data Compression

4.2.4.1 Overview

The ‘glyf’ table contains all the information about simple and composite glyphs in OpenType TrueType font. Simple glyphs are recorded as a collection of data segments representing number of contours, bounding box, TTF instructions for glyph scaling and arrays of data point coordinates. If composite characters are present, they contain the array of simple glyph indexes along with the scaling information and TTF instructions for the composite glyph. These datasets have distinctly different statistical distributions of values and, sometimes, significant levels of redundancy. The use of internal dependencies of glyph data segments and knowledge of TrueType rasterizer behaviour allows the application of additional compression algorithms to reduce glyph data set without any affect on the functionality of glyph rendering and the output quality of TrueType rasterizer.

4.2.4.2 Additional Data Types

TrueType (SFNT) file format often uses USHORT and SHORT data types to provide enough storage space for numerical data. However, in the majority of cases the stored number is rather small and can be recorded within one byte. In order to reduce storage size for TTF glyph data while keeping byte alignment of data records we introduce **255USHORT** and **255SHORT** data type that records the numerical **Value** as follow:

Table 3 — 255USHORT Data Type

Data Type	Syntax	Description and Comments
BYTE	Code	**Value** = Code; /* [0..255] */
	if (Code == 255)	
BYTE	Value1	**Value** = 253 + Value1; /* [253..508] */
	else if (Code == 254)	
BYTE	Value1	**Value** = 506 + Value1; /* [506..761] */
	else if (Code == 253)	
USHORT	Value	**Value**; /* [0..65535] */

Table 4 — 255SHORT Data Type

Data Type	Syntax	Description and Comments
BYTE	Code	Code; /* [0..255] */
	if (Code < 250) {}	**Value** = Code; /* [0..249] */
	else if (Code == 250)	Flip sign code.
BYTE	Value1	**Value** = (SHORT)(–(Value1 + 1));
	else if (Code == 253)	Word data code. Codes 251 and 252 are reserved for “Hop Codes”, see below and subclause 4.2.4.3 for details.
SHORT	Value	**Value**; /* [-32768..32767] */
	else if (Code >= 254)	
BYTE	Value1	**Value =** Value1+256*(255-Code)+250;
	else if (Code==251 || Code==252)	**Value =** Value1+256*(255-Code)+250;
	{}	Reserved **HopCode** values;

4.2.4.3 Glyph Records

4.2.4.3.1 Overview

Glyph records are located in the ‘glyf’ table of OpenType font. Glyphs are recorded as a set of bounding box values, TrueType instructions, flags and outline data points. The glyph record size can be reduced with no effect on the quality of the glyph rendering. This compressed font format converts simple TTF glyph table records to lossless Compact Table Format (CTF).

4.2.4.3.2 Bounding Box

In most cases, the bounding box information can be computed from the data point coordinates and, therefore, bounding box information may be omitted in glyph record.

4.2.4.3.3 Contour End Points

TrueType glyphs store actual numbers of the end point for each contour. CTF stores the number of points in each contour instead, which is typically less then 255. Translation to the endpoint number is simply the cumulative sum of point count in all preceding contours.

4.2.4.3.4 Instructions

TrueType glyphs contain instructions for glyph scaling. Glyph programs typically start with a burst of various PUSH instructions to initialize the stack for a TrueType instructions belonging to that glyph. TrueType provides four different types of PUSH instructions that occupy 18 different opcodes. In order to optimize data storage

size, the CTF format stores only data to be pushed onto the stack. The translation process back to TTF generates a new sequence of optimized TrueType PUSH instructions that results in the identical functionality of glyph program.

The storage format of the initial push data utilizes additional **HopCodes** that make use of the specific data patterns observed in TrueType data. It capitalizes on a specific distribution of the data that a traditional compression technologies do not take advantage of. A typical TrueType data sequence contains value stream that has the same value repeating periodically among other values, e.g. A, x1, A, x2, A, x3, etc. Each value is of type SHORT and occupies 2 bytes of storage. An additional set of HopCodes allows the reduction of the dataset size by applying the following transformation:

- **A, x1, A, x2, A** – translated to: **A, x1, Hop3Code, x2**. (Hop3Code = 0xFB);
- **A, x1, A, x2, A, x3, A** – translated to **A, x1, Hop4Code, x2, x3**, (Hop4Code = 0xFC).

4.2.4.3.5 Glyph Program Table

The following table describes the glyph program in CTF format:

Table 5 — Glyph Program Table

Data Type	Syntax	Description and Comments
255USHORT	pushCount	Number of data items in the push data stream.
255SHORT	pushData[pushCount]	See sample C function for HopCodes processing.
255USHORT	codeSize	Glyph program remaining code size.
BYTE	instructions[codeSize]	Remaining TrueType instructions.

4.2.4.3.6 Translation of HopCodes

The following C code can be used to translate sequence of HopCodes:

```
CHAR *input;
int i;
   BYTE code;
   SHORT A;
SHORT pushCount = Read255UShort(input);
SHORT *pushData = (SHORT *) malloc(sizeof(SHORT) * pushCount);
for ( i = 0; i < pushCount; ) {
    code = *input;
    if (code == Hop3Code ) {
        A = pushData[i-2];
        input++;                    /* Hop3Code */
        pushData[i++] = A;              /* A */
        pushData[i++] = Read255Short(input); /* X2 */
        pushData[i++] = A;              /* A */
    } else if (code == Hop4Code ) {
        A = pushData[i-2];
        input++;                    /* Hop4Code */
        pushData[i++] = A;              /* A */
        pushData[i++] = Read255Short(input); /* X2 */
        pushData[i++] = A;              /* A */
        pushData[i++] = Read255Short(input); /* X3 */
        pushData[i++] = A;              /* A */
    } else {
        pushData[i++] = Read255Short(input); /* Data */
    }
}
```

4.2.4.4 CTF Glyph Data

The following table contains information that describes the glyph data in CTF format.

Table 6 — CTF Glyph Data Table

Data Type	Syntax	Description and Comments
SHORT	numContours	Number of contours in the glyph, also used as a flag.
	if (numContours < 0)	If numContours == -1, the glyph is composite glyph.
	{	
SHORT	xMin	Minimum X value for coordinate data
SHORT	yMin	Minimum Y value for coordinate data
SHORT	xMax	Maximum X value for coordinate data
SHORT	yMax	Maximum Y value for coordinate data
	/* Composite CTF Glyph Table */	
	} else {	
	if (numContours == 0x7FFF)	This is a flag, read new value and bounding box information below.
	{	
SHORT	numContours	Actual number of contours in the glyph.
SHORT	xMin	Minimum X value for coordinate data
SHORT	yMin	Minimum Y value for coordinate data
SHORT	xMax	Maximum X value for coordinate data
SHORT	yMax	Maximum Y value for coordinate data
	} else	Bounding box information omitted and should be computed from the glyph data coordinates.
	{}	
	/* Simple CTF Glyph Table */	
	}	

4.2.4.5 Simple CTF Glyph Data

A simple glyph defines all the contours and points that are used to create the glyph outline. Each point is presented in a (flag, xCoordinate, yCoordinate) triplet that is stored with the variable length encoding consuming 2 to 5 bytes per triplet. The flag value is always stored in the first byte of triplet and defines the format and properties of the point coordinates.

The most significant bit of the flag indicates whether the point is on- or off-curve point. The remaining 7 bits specify 128 possible combinations which define data format and value for X and Y coordinates of a point and specify the combination of the following properties:

- Total number of bytes used for this triplet (byte count);
- Number of bits used for describing X coordinate value (X bits);
- Number of bits for describing Y coordinate value (Y bits);
- An additional amount to be added to X coordinate value (delta X);
- An additional amount to be added to Y coordinate value (delta Y);
- The sign of X coordinate value (X sign);
- The sign of Y coordinate value (Y sign).

Each of the 128 combinations identify a unique set of these seven properties which have been assigned after careful consideration has been given to the typical statistical distribution of data found in TrueType files. As a result, the dataset size in CTF format is significantly reduced compared to the native TrueType format. However, the data is left byte aligned so that an Huffman entropy coder can be easily applied on the second stage.

Unlike the native TrueType formats that allows X and Y coordinates values be defined either in relative or absolute values, the CTF format always defines X and Y coordinates as relative values to the previous point. The first point is relative to (0,0). The information on data formats of xCoordinates[] and yCoordinates[] of glyph data can be found in the subclause 4.2.4.7 of this specification. This table describes simple glyph data in CTF format.

Table 7 — Simple CTF Glyph Table

Data Type	Syntax	Description and Comments
	/* Data */	
255USHORT	contourPoints[numContours]	Number of points in each contour, cumulative sum should be calculated to define total number of points.
BYTE	flags[numPoints]	Array of flags for each coordinate in glyph outline.
Variable length bit field	xCoordinates[numPoints]	Array of X coordinates for glyph outline points.
Variable length bit field	yCoordinates[numPoints]	Array of Y coordinates for glyph outline points.
	/* Glyph Program Table */	

4.2.4.6 Composite CTF Glyph Data

This table describes composite glyph data in CTF format.

Table 8 — Composite CTF Glyph Table

Data Type	Syntax	Description and Comments
	/* Data */	

USHORT	do {
	flags
	. . . . . .
	} while
	(flags & MORE_COMPONENTS);
	if (flags & WE_HAVE_INSTR) {
		/* for all glyph components */
		Flags for each component in composite glyph.
		See “glyf” table specification for more information.

	/* Glyph Program Table */ }	

The information on data formats of flags[], xCoordinates[] and yCoordinates[] of glyph data section can be found in the ‘glyf’ table of OpenType font format specification.

4.2.4.7 Triplet Encoding of X and Y Coordinates

The following table presents information on Triplet Encoding (flag[], xCoordinates[], yCoordinates[]) of simple glyph outline points.

Please note – “Byte Count” field reflects total size of the triplet, including ‘flags’.

Table 9 — Coordinate Encoding Table

Index	Byte Count	X bits	Y bits	Delta X	Delta Y	X sign	Y sign
0	2	8	0	0	N/A	+	N/A
1				0		–	
2				256		+	
3				256		–	
4				512		+	
5				512		–	
6				768		+	
7				768		–	
8				1024		+	
9				1024		–	
10	2	0	8	N/A	0	N/A	+
11					0		–
12					256		+
13					256		–
14					512		+
15					512		–
16					768		+
17					768		–
18					1024		+
19					1024		–
20	2	4	4	1	1	+	+
21					1	+	–
22					1	–	+
23					1	–	–
24					17	+	+
25					17	+	–
26					17	–	+
27					17	–	–
28					33	+	+
29					33	+	–
30					33	–	+
31					33	–	–
32					49	+	+
33					49	+	–
34					49	–	+
35					49	–	–
36	2	4	4	17	1	+	+
37					1	+	–
38					1	–	+
39					1	–	–
40					17	+	+
41					17	+	–
42					17	–	+
43					17	–	–
44					33	+	+
45					33	+	–
46					33	–	+
47					33	–	–
48					49	+	+
49					49	+	–
50					49	–	+
51					49	–	–
52	2	4	4	33	1	+	+
53					1	+	–
54					1	–	+
55					1	–	–
56					17	+	+

Index	Byte Count	X bits	Y bits	Delta X	Delta Y	X sign	Y sign
57					17	+	–
58					17	–	+
59					17	–	–
60					33	+	+
61					33	+	–
62					33	–	+
63					33	–	–
64					49	+	+
65					49	+	–
66					49	–	+
67					49	–	–
68	2	4	4	49	1	+	+
69					1	+	–
70					1	–	+
71					1	–	–
72					17	+	+
73					17	+	–
74					17	–	+
75					17	–	–
76					33	+	+
77					33	+	–
78					33	–	+
79					33	–	–
80					49	+	+
81					49	+	–
82					49	–	+
83					49	–	–
84	3	8	8	1	1	+	+
85					1	+	–
86					1	–	+
87					1	–	–
88					257	+	+
89					257	+	–
90					257	–	+
91					257	–	–
92					513	+	+
93					513	+	–
94					513	–	+
95					513	–	–
96	3	8	8	257	1	+	+
97					1	+	–
98					1	–	+
99					1	–	–
100					257	+	+
101					257	+	–
102					257	–	+
103					257	–	–
104					513	+	+
105					513	+	–
106					513	–	+
107					513	–	–
108	3	8	8	513	1	+	+
109					1	+	–
110					1	–	+
111					1	–	–
112					257	+	+
113					257	+	–
114					257	–	+

Index	Byte Count	X bits	Y bits	Delta X	Delta Y	X sign	Y sign
115					257	–	–
116					513	+	+
117					513	+	–
118					513	–	+
119					513	–	–
120	4	12	12	0	0	+	+
121						+	–
122						–	+
123						–	–
124	5	16	16	0	0	+	+
125						+	–
126						–	+
127						–	–

The coordinate values of simple glyph outline points are calculated as follows:

```
Xcoord[i] = (SHORT)((Xsign)*(xCoordinate[i] + DeltaX[index]);
Ycoord[i] = (SHORT)((Ysign)*(yCoordinate[i] + DeltaY[index]);
```

5 Font Data Stream

5.1 Structure of the Font Data Stream

Font data streams are streams of streamType FontDataStream 0x0C (see ISO/IEC 14496-1, Table 7), identified by a specific objectTypeIndication 0x06 (see ISO/IEC 14496-1, Table 6). A font data stream consists of one or more access units with font data.

5.2 Access Unit Definition

5.2.1 Overview

A font data access unit conveys data for one single font, or a subset thereof. Two access unit formats are specified, discriminated by the DecoderSpecificInfo for the font data stream (see subclause 5.4). Only one of these access unit formats can be selected for a given font data stream.

The basic access unit format permits to convey font data without any header information. The DecoderSpecificInfo provides the identification of the font and, possibly, the subset thereof that is conveyed. The content of subsequent basic access units in such a font data stream shall replace the font or font subset data of previous basic access units.

The enhanced access unit format provides a self-contained format where an access unit header contains all necessary information about the font data conveyed in this access unit.

Access units in font data streams shall be labelled and time-stamped by suitable means. This shall be done via the related flags and the composition time stamps, respectively, in the SL packet header (see ISO/IEC 14496-1, 10.2.4). The composition time indicates the point in time at which a font data access unit becomes valid, i.e., when the embedded font information becomes available for font rendering. Decoding and composition time for a font data access unit shall always have the same value.

For all basic access units and those enhanced access units that have `fontSubsetExtensionFlag` set to '0' the `randomAccessPointFlag` in the SL packet header shall be set to '1' for this access unit. Otherwise, the `randomAccessPointFlag` shall be set to '0'.

5.2.2 Syntax

```
class BasicFontAccessUnit() {
   bit(8)  fontData[sizeOfInstance];
}

class EnhancedFontAccessUnit() {
   bit(7)  fontFormat;
   bit(1)  storeFont;
   bit(8)  fontNameLength;
   bit(8)  fontName[fontNameLength];
   bit(7)  fontSubsetID;
   bit(1)  fontSubsetExtensionFlag;
   bit(8)  fontSpecInfoLength;
   bit(8)  fontSpecInfo[fontSpecInfoLength];
   bit(8)  fontData[sizeOfInstance - fontNameLength - fontSpecInfoLength - 5]
}
```

5.2.3 Semantics

`fontFormat` – indicates the format of the access unit as follows.

Table 10 — `fontFormat` values for EnhancedAccessUnit

`fontFormat`	Access unit content
0x00	Forbidden
0x01	OpenType with TrueType outlines (uncompressed)
0x02	OpenType with TrueType outlines compressed using the compression mechanism described in subclause 4.2
0x03	OpenType CFF
0x04 – 0x3F	ISO reserved
0x40 – 0x7F	User private

`storeFont` – if set to '1', the terminal shall store the font information contained within the access unit at least until the end of the current session. Otherwise, the font information becomes unavailable as soon as the font data stream becomes unavailable.

NOTE — Details of persistent storage are beyond the scope of this specification.

`fontNameLength` – specifies the length in characters of the `fontName` field.

`fontName` – a Unicode (ISO/IEC 10646-1) encoded string that indicates the name of the font.

`fontSubsetID` – specifies a numerical ID for a subset of the font conveyed in this access unit. If `fontSubsetID` equals 0x00, the access unit contains the complete font data. The value of 0x7F shall not be used.

`fontSubsetExtensionFlag` – if set to one indicates that the font data in this access unit is an extension of the previously sent font subset with the same `fontName` and `fontSubsetID`. Otherwise the font subset replaces any previous font subset with the same `fontName` and `fontSubsetID`.

`fontSpecInfoLength` - defines the length in bytes of the `fontSpecInfo` field.

`fontSpecInfo` – is an opaque container with information for a specific font handler.

`fontData` – contains the font data in the specified `fontFormat`.

5.3 Time Base for Font Data Streams

The time base associated with a font data stream shall be indicated by suitable means. This shall be done by the use of object clock reference time stamps in the SL packet headers (see ISO/IEC 14496-1, 10.2.4) for this stream or by indicating the elementary stream from which this font data stream inherits the time base (see ISO/IEC 14496-1, 10.2.3). All time stamps in the SL-packetized font data stream refer to this time base.

5.4 Font Data Decoder Configuration

5.4.1 Syntax

```
class FontDataDecoderConfiguration extends DecoderSpecificInfo  : bit(8)
tag=DecSpecificInfoTag {
   bit(7)  fontFormat;
   if (fontFormat != 0x00) {
      bit(1)  storeFont;
      bit(8)  fontNameLength;
      bit(8)  fontName[fontNameLength];
      bit(7)  fontSubsetID;
      bit(1)  reserved = 1;
      bit(8)  fontSpecInfo[sizeOfInstance - fontNameLength - 4];
   }
}
```

5.4.2 Semantics

`fontFormat` – indicates the format of the access unit as follows.

Table 11 — `fontFormat` values for FontDataDecoderConfiguration

`fontFormat`	Access unit content
0x00	enhanced access unit format
0x01	OpenType with TrueType outlines (uncompressed)
0x02	OpenType with TrueType outlines compressed using the compression mechanism described in subclause 4.2
0x03	OpenType CFF
0x04 – 0x3F	ISO reserved
0x40 – 0x7F	User private

A `fontFormat` of 0x00 indicates the use of the enhanced access unit format for this stream. In this format, all information about the font data is conveyed within each access unit.

`storeFont` – if set to '1', the terminal shall store the font information contained within the access unit at least until the end of the current session. Otherwise, the font information becomes unavailable as soon as the font data stream becomes unavailable.

NOTE — Details of persistent storage are beyond the scope of this specification.

`fontNameLength` – specifies the length in characters of the `fontName` field.

`fontName` – a Unicode (ISO/IEC 10646-1) encoded string that indicates the name of the font.

`fontSubsetID` – specifies a numerical ID for a subset of the font conveyed in this access unit. If `fontSubsetID` equals 0x00, the access unit contains the complete font data. The value of 0x7F shall not be used.

`fontSpecInfoLength` - defines the length in bytes of the `fontSpecInfo` field.

`fontSpecInfo` – is an opaque container with information for a specific font handler.