



DPC: 25/30479034 DC

BSI Group Headquarters

389 Chiswick High Road London W4 4AL

Tel: +44 (0)20 8996 9000

Fax: +44 (0)20 8996 7400

www.bsigroup.com

Date: 24 February 2025

Origin: International

Latest date for receipt of comments: 19 April 2025

Project No. 2023/01952

Responsible committee: IST/37 Coding of picture, audio, multimedia and hypermedia information

Interested committees:

Title: Draft BS ISO/IEC 14496-22 Information technology — Coding of audio-visual objects. Part 22: Open font format.

Please notify admin.start@bsigroup.com if you are aware of any keywords that might assist in clarifying or identifying the standard or of the content of this standard:

- i) has any issues related to 3rd party IPR, patent or copyright
- ii) affects other national standard(s)
- iii) requires additional national guidance or information

WARNING: THIS IS A DRAFT AND MUST NOT BE REGARDED OR USED AS A BRITISH STANDARD. THIS DRAFT IS NOT CURRENT BEYOND 19 April 2025This draft is issued to allow comments from interested parties; all comments will be given consideration prior to publication. No acknowledgement will normally be sent. **See overleaf for information on the submission of comments.**

No copying is allowed, in any form, without prior written permission from BSI except as permitted under the Copyright, Designs and Patent Act 1988 or for circulation within a nominating organization for briefing purposes. Electronic circulation is limited to dissemination by e-mail within such an organization by committee members.

Further copies of this draft may be purchased from BSI Shop <http://shop.bsigroup.com> or from BSI Customer Services, Tel: +44(0) 20 8996 9001 or email cservices@bsigroup.com. British, International and foreign standards are also available from BSI Customer Services.Information on the co-operating organizations represented on the committees referenced above may be obtained from <https://standardsdevelopment.bsigroup.com/>.E-mail: admin.start@bsigroup.com

Introduction

This draft standard is based on national and international discussions. Your comments on this draft are invited and will assist in the preparation of the consequent standard.

For international standards, comments will be reviewed by the relevant UK national committee before sending the consensus UK vote and comments to the international committee, which will then decide appropriate action. If the international standard is approved, it is usual for the text to be published as a British Standard.

For national standards, comments will be reviewed by the relevant UK national committee and the resulting standards published as a British Standard.

UK Vote

Please indicate whether you consider the UK should submit a negative (with supporting technical reasons) or positive vote on this draft. Please indicate if you are aware of any reason why this draft standard should not be published as a British Standard.

Submission of Comments

- **Annotated drafts are not acceptable and will be rejected.**
- All comments should be submitted online at <https://standardsdevelopment.bsigroup.com/>. You will need to register in order to comment.

Template for comments and secretariat observations

Date: xx/xx/20xx	Document: ISO/DIS xxxx
------------------	------------------------

1	2	(3)	4	5	(6)	7
MB	Clause No./ Subclause No./Annex (e.g. 3.1)	Paragraph/ Figure/Table/Note	Type of comment	Comment (justification for change) by the MB	Proposed change by the MB	Secretariat observations on each comment submitted
	3.1	Definition 1	ed	Definition is ambiguous and needs clarifying.	Amend to read '...so that the mains connector to which no connection...'	
	6.4	Paragraph 2	te	The use of the UV photometer as an alternative cannot be supported as serious problems have been encountered in its use in the UK.	Delete reference to UV photometer.	



DRAFT International Standard

ISO/IEC DIS 14496-22

Information technology — Coding of audio-visual objects —

Part 22: Open font format

ICS: 35.040.40

ISO/IEC JTC 1/SC 29

Secretariat: JISC

Voting begins on:
2025-02-24

Voting terminates on:
2025-05-19

This document is circulated as received from the committee secretariat.

THIS DOCUMENT IS A DRAFT CIRCULATED FOR COMMENTS AND APPROVAL. IT IS THEREFORE SUBJECT TO CHANGE AND MAY NOT BE REFERRED TO AS AN INTERNATIONAL STANDARD UNTIL PUBLISHED AS SUCH.

IN ADDITION TO THEIR EVALUATION AS BEING ACCEPTABLE FOR INDUSTRIAL, TECHNOLOGICAL, COMMERCIAL AND USER PURPOSES, DRAFT INTERNATIONAL STANDARDS MAY ON OCCASION HAVE TO BE CONSIDERED IN THE LIGHT OF THEIR POTENTIAL TO BECOME STANDARDS TO WHICH REFERENCE MAY BE MADE IN NATIONAL REGULATIONS.

RECIPIENTS OF THIS DRAFT ARE INVITED TO SUBMIT, WITH THEIR COMMENTS, NOTIFICATION OF ANY RELEVANT PATENT RIGHTS OF WHICH THEY ARE AWARE AND TO PROVIDE SUPPORTING DOCUMENTATION.

Reference number
ISO/IEC DIS 14496-22:2025(en)

Full document details: 25/30479034 DC

© ISO/IEC 2025  **Documents**
Document Sharing Hub



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2025

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword.....	viii
Introduction.....	ix
1 Scope.....	1
2 Normative references.....	1
3 Terms, definitions and abbreviated terms.....	1
3.1 Terms and definitions	1
3.2 Abbreviated terms.....	2
4 The Open Font file format.....	3
4.1 Description.....	3
4.2 Filenames.....	3
4.3 Data types	3
4.4 Table version numbers.....	7
4.5 Top-level OFF organization	8
4.5.1 Table directory	8
4.5.2 Table records	9
4.5.3 Calculating checksums.....	10
4.6 Font collections.....	11
4.6.1 The Font Collection overview.....	11
4.6.2 The Font Collection file structure	11
4.6.3 TTC header	12
5 Open font tables.....	14
5.1 Required common tables	14
5.1.1 List of required tables.....	14
5.1.2 cmap—Character to glyph index mapping table	15
5.1.3 head—Font header.....	31
5.1.4 hhea—Horizontal header	35
5.1.5 hmtx—Horizontal metrics.....	36
5.1.6 maxp—Maximum profile	38
5.1.7 name—Naming table	39
5.1.8 OS/2 — OS/2 and Windows metrics table	53
5.1.9 post—PostScript	82
5.1.10 HHEA—Horizontal header	86
5.1.11 HMTX—Horizontal metrics.....	87
5.1.12 MAXP—Maximum Profile, for 24-bit fonts	89
5.2 Tables related to TrueType outlines	90
5.2.1 List of TrueType outlines tables.....	90
5.2.2 cvt—Control value table.....	91
5.2.3 fpgm—Font program	91
5.2.4 glyf—Glyf data.....	91
5.2.5 loca—Index to location.....	100
5.2.6 prep—Control value program.....	100
5.2.7 gasp—Grid-fitting and scan-conversion procedure table.....	101
5.2.8 GLYPH—Glyph data	103
5.2.9 LOCA—Index to location	113
5.3 Tables related to CFF outlines.....	113

5.3.1	List of CFF outline tables.....	113
5.3.2	CFF—Compact Font Format table (version 1)	114
5.3.3	CFF2—Compact Font Format (version 2) table.....	114
5.3.4	CFSH— Compact Font Format supplementary hint table	174
5.3.5	VORG—Vertical origin table.....	175
5.4	SVG—The SVG (Scalable Vector Graphics) table	176
5.4.1	Introduction	176
5.4.2	SVG table formats.....	177
5.4.3	SVG documents.....	178
5.4.4	Color and color palettes.....	180
5.4.5	Glyph Identifiers.....	182
5.4.6	Glyph semantics and text layout processing	183
5.4.7	Coordinate Systems and Glyph Metrics.....	183
5.4.8	Animations	184
5.4.9	SVG glyph examples.....	184
5.5	Tables related to bitmap glyphs.....	191
5.5.1	List of bitmap glyph tables.....	191
5.5.2	EBDT—Embedded bitmap data table	191
5.5.3	EBLC—Embedded bitmap location table	196
5.5.4	EBSC—Embedded bitmap scaling table	204
5.5.5	CBDT—Color bitmap data table	205
5.5.6	CBLC—Color bitmap location table	207
5.5.7	sbix—Standard bitmap graphics table	209
5.6	Optional tables	212
5.6.1	DSIG—Digital signature table	212
5.6.2	hdmx—Horizontal device metrics	215
5.6.3	kern—Kerning	216
5.6.4	LTSH—Linear threshold	220
5.6.5	MERG—Merge table.....	220
5.6.6	meta—Metadata table	225
5.6.7	PCLT—PCL 5 table.....	230
5.6.8	VDMX—Vertical device metrics	237
5.6.9	vhea—Vertical header table.....	240
5.6.10	vmtx—Vertical metric table	245
5.6.11	COLR—Color Table	247
5.6.12	CPAL—Palette Table	316
5.6.13	VHEA—Vertical header table.....	321
5.6.14	VMTX—Vertical metric table	326
5.6.15	DMAP—Delta map table	327
6	Open Font advanced layout tables	328
6.1	Open Font advanced layout extensions.....	328
6.1.1	Overview of advanced typographic layout extensions	328
6.1.2	OFF Layout terminology	330
6.1.3	Text processing with OFF layout	333
6.1.4	OFF layout and Font variations	334
6.2	OFF Layout common table formats	335
6.2.1	Overview	335
6.2.2	OFF Layout and Font variations	336
6.2.3	Table organization.....	337
6.2.4	Scripts and languages	338
6.2.5	Features and lookups	341
6.2.6	Coverage table.....	348
6.2.7	Class definition table.....	350

6.2.8	Common formats for contextual lookup subtables	352
6.2.9	Device and VariationIndex tables	370
6.2.10	Conditions and Condition Sets	372
6.2.11	Feature variations	379
6.2.12	Common table examples	383
6.3	Advanced typographic tables	392
6.3.1	BASE—Baseline table	392
6.3.2	GDEF—The glyph definition table	416
6.3.3	GPOS—The glyph positioning table	431
6.3.4	GSUB—The glyph substitution table	493
6.3.5	JSTF—The justification table	529
6.3.6	MATH—The mathematical typesetting table	542
6.4	Layout tag registry	562
6.4.1	Overview	562
6.4.2	Scripts tags	562
6.4.3	Language tags	569
6.4.4	Feature tags	596
6.4.5	Baseline tags	678
7	OFF font variations	683
7.1	Font variations overview	683
7.1.1	General	683
7.1.2	Terminology	686
7.1.3	Variation space, default instances and adjustment deltas	688
7.1.4	Coordinate scales and normalization	691
7.1.5	Variation data	696
7.1.6	Variation data tables and miscellaneous requirements	705
7.1.7	Algorithm for interpolation of instance values	707
7.1.8	Interpolation example	711
7.1.9	Dynamic generation of static instance fonts	718
7.2	Font variations common table formats	719
7.2.1	Overview	719
7.2.2	Tuple variation store	720
7.2.3	Item variation stores	729
7.2.4	Design-variation axis tag registry	739
7.3	Font variations tables	746
7.3.1	avar—Axis variations table	746
7.3.2	cvar—CVT variations table	751
7.3.3	fvar—Font variations table	753
7.3.4	gvar—Glyph variations table	763
7.3.5	HVAR—Horizontal metrics variations table	774
7.3.6	MVAR—Metrics variations table	777
7.3.7	STAT—Style attributes table	782
7.3.8	VVAR—Vertical metrics variations table	798
7.3.9	GVAR—Glyph variations table	799
7.3.10	VARC—Variable Composite Glyph Descriptions	811
8	Recommendations for OFF fonts	818
8.1	Byte ordering	819
8.2	Mixing outline formats	819
8.3	Filenames	819
8.4	Table alignment and length	819
8.5	Glyph 0: the .notdef glyph	819
8.6	'BASE' table	820
8.7	'cmap' table	820

8.8 'cvt' table.....	820
8.9 'fpgm' table	821
8.10 'glyf' table.....	821
8.11 'hdmx' table.....	821
8.12 'head' table	821
8.13 'hhea' table	821
8.14 'hmtx' table.....	821
8.15 'kern' table.....	822
8.16 'loca' table.....	822
8.17 'LTSH' table.....	822
8.18 'maxp' table	822
8.19 'name' table	822
8.20 'OS/2' table	825
8.21 'post' table.....	826
8.22 'prep' table.....	826
8.23 'VDMX' table	826
8.24 TrueType Collections.....	826
9 General recommendations.....	826
9.1 Optimized table ordering	826
9.2 Non-standard (Symbol) fonts.....	827
9.3 Baseline to baseline distances	827
9.4 Style bits.....	828
9.5 Drop-out control	828
9.6 Embedded bitmaps	829
9.7 OFF CJK font guidelines	829
9.8 Stroke reduction in variable fonts	829
9.9 Families with optical size variants.....	830
Annex A (informative) IBM font family classifications	831
A.1 General	831
A.2 Class ID=0 No Classification.....	831
A.3 Class ID=1 Oldstyle Serifs	831
A.4 Class ID=2 Transitional Serifs.....	832
A.5 Class ID=3 Modern Serifs.....	833
A.6 Class ID=4 Clarendon Serifs.....	834
A.7 Class ID=5 Slab Serifs	835
A.8 Class ID=6 (reserved for future use)	836
A.9 Class ID=7 Freeform Serifs.....	836
A.10 Class ID=8 Sans Serifs	837
A.11 Class ID=9 Ornaments.....	838
A.12 Class ID=10 Scripts	839
A.13 Class ID=11 (reserved for future use)	841
A.14 Class ID=12 Symbolic.....	841
A.15 Class ID=13 Reserved	842
A.16 Class ID=14 Reserved	842
Annex B (informative) Earlier versions of OS/2 – OS/2 and Windows metrics.....	843
B.1 OS/2 - OS/2 and Windows metrics (version 0).....	843
B.2 OS/2 - OS/2 and Windows metrics (version 1).....	863
B.3 OS/2 - OS/2 and Windows metrics (version 2).....	886
B.4 OS/2 - OS/2 and Windows metrics (version 3).....	908
B.5 OS/2 - OS/2 and Windows metrics (version 4).....	929
Annex C (informative) OFF Mirroring Pairs List	953

Annex D (informative) Comparison of 'glyf', 'CFF ' and 'CFF2' tables.....	960
Annex E (informative) Registration of Media Type: application/font-sfnt.....	962

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

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. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see <http://patents.iec.ch>).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

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

This fifth edition cancels and replaces the fourth edition (ISO/IEC 14496-22:2019), which has been technically revised. It also incorporates the Amendments ISO/IEC 14496-22:2019/Amd.1:2020 and ISO/IEC 14496-22:2019/Amd.2:2022.

The main changes compared to the previous edition are as follows:

- new technology clauses were added;
- many existing clauses, subclauses, tables, figures and annexes were editorially revised.

A list of all parts in the ISO/IEC 14496 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

Multimedia applications require a broad range of media-related standards. In addition to the typical audio and video applications, multimedia presentations include scalable 2D graphics and text supporting all languages of the world. Faithful reproduction of scalable multimedia content requires additional components including scalable font technology. The Open Font Format is a widely-supported format for font data with a rich set of capabilities for digital typography. It is based on the OpenType®¹ font format, which was originally developed as an extension of the TrueType™² font format, using the same 'sfnt' container structure, and maintaining compatibility for fonts created following the original TrueType specification. But several additional capabilities are also supported, including the following:

- Glyph outline data can use the CFF or CFF version 2 ("CFF2") formats, as well as the TrueType glyph format.
- Multicolour glyph presentation is supported using embedded colour bitmaps or SVG documents, or using 2D graphic compositions defined in the font using a binary format that combines outline-format glyphs with various graphics operations.
- All Unicode® characters can be supported, including supplementary-plane characters, as well as Unicode variation sequences.
- Advanced Open Font Layout tables provide the advanced typographic capabilities needed for high-quality typography as well as for international text using the wide variety of scripts supported in The Unicode Standard®.
- The mathematical typesetting table allows a font to include data required for layout of complex, math formulas.
- Open Font collection files enable multiple fonts that share common data to be housed within a single file, allowing for de-duplication of data. This is especially useful, for example, for sets of CJK (Chinese, Japanese, Korean) fonts of the same design that share most glyphs in common but that vary with locale-specific glyphs for certain characters.
- Font variations ("variable fonts") enable glyph outlines or other font data to be variable based on one or more design-axis parameters. Whereas a collection file can contain multiple discrete, static font resources, a variable font can provide continuous variation in design along each of its axes. This can provide great flexibility for content authors and designers while also allowing the font data for an entire font family to be represented in an efficient format.

This specification is intended to be used in conjunction with other specifications.

- While various legacy character encoding standards are supported, it is primarily designed for use with The Unicode Standard, which provides the universal encoding for written characters and symbols, as well as specifications for how text in different scripts is to be represented.

¹ OpenType® is the registered trademark of a product supplied by Microsoft. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

² TrueType™ is the trademark of a product supplied by Apple Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

- This specification defines Advanced Open Font Layout tables and low-level glyph substitution and positioning operations needed for high-quality typography and for correct display of Unicode text in various scripts. (See 6.1.1.) The OpenType Layout feature registry defines various features that represent specific typographic capabilities that can be supported in a font, and that are used to activate those capabilities in a given font. Many features expose optional capabilities that authors and typographers can choose to use at their discretion; for example, small cap forms, or kerning. But many other features are used to activate capabilities that are required for correct display of text; for example, required ligatures for Arabic script, or positioning of mark glyphs. Many scripts supported in Unicode have complex structural behaviors that require non-trivial operations, implemented in applications or in text layout and “shaping” libraries, to derive a correct sequence of positioned glyphs for presentation of an underlying Unicode string. The feature registry will include features that can be used in those operations. However, complete specification of shaping algorithms for different scripts is beyond the scope of this specification. Such algorithms can be proprietary, “closed-source” implementations in particular applications; or they can be defined in vendor-specific specifications or in other industry specifications.
- Support for mathematical text involving complex formulas requires a content format to describe text semantics combined with layout and presentation capabilities. This specification defines the font-specific data that would be needed for presentation. In this way, the Open Font Format can be used to implement presentation capabilities for other document format specifications, such as T_EX or MathML. Certain features can be defined in the Advanced Open Font Layout feature tag registry to support math layout operations.
- Text layout involves operations within individual lines of text but also control over arrangement of lines into larger blocks within a page or similar context. This specification defines certain data that is used in block-level layout, such as default line metrics (ascent, descent, leading). Also, certain features can be defined in the Advanced Open Font Layout feature tag registry to support layout of text blocks for either horizontal or vertical layout orientation. Complete specification of block level layout is beyond the scope of this specification, however. It can be used in conjunction with other specifications, such as Unicode Standard Annex #50: Unicode Vertical Text Layout.

Some font capabilities can be subject to tailoring by applications or other higher-level protocols. For example, the TrueType instructions defined in this specification define operations used in rasterization of glyph outlines, but applications can supplement their own algorithms for final rasterization (for example, using over-sampling, or control of sub-pixel display elements) to optimize legibility of glyphs. Also, while the descriptions of features in the Advanced Open Font Layout feature tag registry can specify expected usage, applications may tailor their usage of features subject to other specifications or according to their own needs.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of patents.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights. The holders of these patent rights have assured ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holders of these patent rights is registered with ISO and IEC.

Information may be obtained from:

Apple Inc.
1 Infinite Loop MS 3-PAT

US-Cupertino, CA 95014-2084
Tel.: +1 408 974 9453
Email: iplaw@apple.com

Microsoft Corporation
Interoperability Group 3460 157th Avenue NE
US-Redmond, WA 98052
Tel.: +1 425 882 80 80

Monotype
600 Unicorn Park Drive
US-Woburn, MA 01801
Tel.: +1 781-970-6000

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

Information technology — Coding of audio-visual objects —

Part 22:

Open font format

1 Scope

This document specifies the Open Font Format (OFF) specification, including the TrueType and Compact Font Format (CFF) outline formats. Many references to both TrueType and PostScript exist throughout this document, as Open Font Format fonts combine the two technologies. The document defines data structures for various font tables, and provides the necessary details for developers to build a font rendering and text layout/shaping engines in compliance with this document.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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 10646, Information technology — Universal Coded Character Set (UCS)

ISO/IEC 14496-18, Information technology — Coding of audio-visual objects — Part 18: Font compression and streaming

ISO/IEC 15948, Information technology — Computer graphics and image processing — Portable Network Graphics: Functional specification³

IEC 61966-2-1/Amd 1:2003: Multimedia systems and equipment — Colour measurement and management — Part 2-1: Colour management — Default RGB colour space — sRGB.

TrueType Instruction Set, <http://www.microsoft.com/typography/otspec/ttinst.htm>

The Unicode Standard, Version 15.1.0, <http://www.unicode.org/versions/Unicode15.1.0/>

Scalable Vector Graphics (SVG) 1.1 (Second edition), W3C Recommendation, 16 August 2011 <http://www.w3.org/TR/SVG11/>

IETF BCP 47 specification, “Tags for Identifying Languages”. <https://www.rfc-editor.org/info/bcp47>

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

“Engine” – a software implementation of a technology.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

³ Also available as a W3C Recommendation (Reference [15]).

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <http://www.electropedia.org/>

3.2 Abbreviated terms

ACF	average character face
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
BMP	[Unicode] basic multilingual plane
BTBD	baseline to baseline distance
CFF	compact font format
CID	character identifier
CJK	Chinese Japanese Korean [characters, ideographs, fonts, etc.]
CJKV	Chinese Japanese Korean and Vietnamese
CV	control value
CVT	control value table
DLL	dynamically linked library
FDEF	function definition
GID	glyph ID
ICF	ideographic character face
IDEF	instruction definition
IETF	Internet Engineering Task Force
JIS	Japanese Industrial Standard
LTR	left to right
NLC	National Language Council of Japan
OFF	open font format
OMPL	OFF mirroring pairs list
OTF	OpenType font
PCL	printer control language
PPM, PPEM	pixels per em
RTL	right to left
TTC	TrueType collection
TTF	TrueType font
UCS	universal character set
UTF	Unicode transformation format
UVS	Unicode variation sequence
VM	virtual memory
W3C	world wide web consortium

4 The Open Font file format

4.1 Description

An Open Font file contains data, in table format, used for rendering of text. Portions of the data are used by applications to calculate the layout of text using the font; that is, the selection of glyphs and their placement within a line. Other data provide descriptions of glyphs as TrueType or Compact Font Format (CFF) outlines. Still other data can provide monochromatic or colour bitmaps, SVG documents or binary, 2D vector graphics compositions as alternate glyph descriptions. The font data also includes meta-information, such as name strings that can be used to present the font as an available choice in a font-picker user interface. Each of these types of data is stored in one or more tables each designed for a particular purpose.

References to the Universal Coded Character Set and the Unicode Standard are used throughout this document; the users of the OFF cannot meet the requirements of this document without strict adherence to these standards.

4.2 Filenames

OFF font files may have the extension .OTF, .TTF, .OTC or .TTC. (The extension may be upper or lower case.) The extensions .OTC and .TTC should only be used for font collection files. For additional information on filename extension conventions, see [subclause 8.3](#).

4.3 Data types

The following data types are used in the OFF font file. All OFF fonts use big-endian (network) byte order:

Data Type	Description
uint8	8-bit unsigned integer.
int8	8-bit signed integer.
uint16	16-bit unsigned integer.
int16	16-bit signed integer.
uint24	24-bit unsigned integer.
uint32	32-bit unsigned integer.
int32	32-bit signed integer.
uint32var	Encodes a 32-bit integer (uint32) in a variable number of bytes.
Fixed	32-bit signed fixed-point number (16.16)
FWORD	int16 that describes a quantity in font design units.
UFWORD	uint16 that describes a quantity in font design units.
F2DOT14	16-bit signed fixed number with the low 14 bits of fraction (2.14).

Data Type	Description
F4DOT12	16-bit signed fixed number with the low 12 bits of fraction (4.12).
F6DOT10	16-bit signed fixed number with the low 10 bits of fraction (6.10).
LONGDATETIME	Date and time represented in number of seconds since 12:00 midnight, January 1, 1904, UTC. The value is represented as a signed 64-bit integer.
Tag	Array of four uint8s (length = 32 bits) used to identify a table, design-variation axis, script, language system, feature, or baseline.
TupleValues	A version of Packed Deltas supporting 32-bit values.
Offset8	8-bit offset to a table, same as uint8, NULL offset = 0x00
Offset16	Short offset to a table, same as uint16, NULL offset = 0x0000
Offset24	24-bit offset to a table, same as uint24, NULL offset = 0x000000
Offset32	Long offset to a table, same as uint32, NULL offset = 0x00000000
Version16Dot16	Packed 32-bit value with major and minor version numbers. (See 4.4.)

The F2DOT14 format consists of a signed, 2's complement integer and an unsigned fraction. To compute the actual value, take the integer and add the fraction. Examples of 2.14 values are:

Decimal Value	Hex Value	Integer	Fraction
1.999939	0x7fff	1	16383/16384
1.75	0x7000	1	12288/16384
0.000061	0x0001	0	1/16384
0.0	0x0000	0	0/16384
-0.000061	0xffff	-1	16383/16384
-2.0	0x8000	-2	0/16384

The F4DOT12 and F6DOT10 formats work the same way as F2DOT14 but have a larger integer and less precision in the unsigned fraction part.

A Tag value is a uint8 array. Each byte within the array shall have a value in the range 0x20 to 0x7E. This corresponds to the range of values of Unicode Basic Latin characters in UTF-8 encoding, which is the same as the printable ASCII characters. As a result, a Tag value can be re-interpreted as a four-character sequence, which is conventionally how they are referred to. Formally, however, the value is a byte array. When re-interpreted as characters, the Tag value is case sensitive. It shall have one to four non-space characters, padded with trailing spaces (byte value 0x20). A space character shall not be followed by a non-space character.

The **uint32var** format is an efficient and compact encoding of uint32 values, using from one to five bytes to store a single value.

Values are decoded as follows:

If the current byte's most significant bit (0x80) is zero, return the value.

Otherwise, the most significant bits are taken to indicate the number of bytes to follow, as indicated in the following table; the values are stored with the first byte being the most significant.

Bits				Meaning
0x80 (1000 0000)	0x40 (0100 0000)	0x20 (0010 0000)	0x10 (0001 0000)	
zero	(any value)	(any value)	(any value)	All 7 remaining bits are the value
1	0	(any value)	(any value)	One byte follows; the low 6 bits of the first are shifted left 8 bits (multiplied by 256) and the single following byte is added.
1	1	0	(any value)	Two bytes follow; the low 5 bits of the first are shifted left by 16 bits and added, then the following byte is shifted left by 8 bits and added (bitwise "or"), and then the second following byte is added.
1	1	1	0	Three bytes follow; the low 4 bits of the first byte become the most significant bits of the new value, as above.
1	1	1	1	Four bytes follow; the low 4 bits of the first byte become the most significant bits of the new value, as above. The least significant 4 bits of the initial byte shall be zero in this case to prevent overflow.

The following pseudo-code can be adapted to read and write these variable-byte numbers. Note that although they were inspired by UTF-8 encoding, they are not the same, and achieve greater compression in the font encoding context.

```
def _readuint32var(data, i):
    """Read a variable-length number from data starting at index i.
    Return the number and the next index.
    """

    b0 = data[i]
    if b0 < 0x80:
        return b0, i + 1
    elif b0 < 0xC0:
        return (b0 - 0x80) << 8 | data[i + 1], i + 2
    elif b0 < 0xE0:
        return (b0 - 0xC0) << 16 | data[i + 1] << 8 | data[i + 2], i + 3
    elif b0 < 0xF0:
```

```

        return (b0 - 0xE0) << 24 | data[i + 1] << 16 | data[i + 2] << 8 | data[
            i + 3
        ], i + 4
    else:
        return (b0 - 0xF0) << 32 | data[i + 1] << 24 | data[i + 2] << 16 | data[
            i + 3
        ] << 8 | data[i + 4], i + 5

def _writeuint32var(v):
    """Write a variable-length number.
    Return the data.
    """
    if v < 0x80:
        return struct.pack(">B", v)
    elif v < 0x4000:
        return struct.pack(">H", (v | 0x8000))
    elif v < 0x200000:
        return struct.pack(">L", (v | 0xC00000))[1:]
    elif v < 0x10000000:
        return struct.pack(">L", (v | 0xE0000000))
    else:
        return struct.pack(">B", 0xF0) + struct.pack(">L", v)

```

The **TupleValues** format is essentially the same as Packed Deltas (see 7.2.2.6) from the TupleVariationStore, but modified to allow storing 32-bit values: if DELTAS_ARE_ZERO and DELTAS_ARE_WORDS are both set in the control byte, then the following values are interpreted as 32-bit.

NOTE Those two flags, corresponding to the topmost two bits in the control byte, could not otherwise both be set.

TupleValues are used in two ways:

1. When the number of values to be decoded is known in advance, decoding stops when all the values are decoded.
2. When the number of values to be decoded is not known in advance, but the number of encoded bytes is known, then values are decoded until all the encoded bytes have been processed.

Within this specification, many structures are defined in terms of the data types listed above. Structures are characterized as either *records* or *tables*. The distinction between records and tables is based on these general criteria:

- Tables are referenced by offsets. If a table contains an offset to a sub-structure, the offset is normally from the start of that table.
- Records occur sequentially within a parent structure, either within a sequence of table fields or within an array of records of a given type. If a record contains an offset to a sub-structure, that structure is logically a subtable of the record's parent table and the offset is normally from the start of the parent table.

In some cases, fields for subtable offsets are documented as permitting NULL values when the given subtable is optional. For example, in the 'BASE' table header, the horizAxisOffset and vertAxisOffset fields may be NULL, meaning that either subtable (or both) is optional. A NULL subtable offset always indicates that the given subtable is not present. Applications shall never interpret a NULL offset value as the offset to subtable data. For cases in which subtable offset fields are not documented as permitting

NULL values, font compilers shall include a subtable of the indicated format, even if it is a header stub without further data (for example, a coverage table with no glyph IDs). Applications parsing font data should, however, anticipate non-conformant font data that has a NULL subtable offset where only a non-NULL value is expected.

4.4 Table version numbers

Most tables have version numbers, and the version number for the entire font is contained in the table directory. Note that there are five different table version number types, each with its own numbering scheme.

- A single uint16 field. This is used in a number of tables, usually with versions starting at zero (0).
- Separate, uint16 major and minor version fields. This is used in a number of tables, usually with versions starting at 1.0.
- A uint32 field with enumerated values.
- A uint32 field with a numeric value. This is used only in the 'DSIG' and 'meta' tables.
- A Version16Dot16 field for major/minor version numbers. This is used only in the 'maxp', 'post' and 'vhea' tables.

The Version16Dot16 type is used for the version field of certain tables, and only for reasons of backward compatibility. (In earlier editions, these fields were documented as using a Fixed value, but had minor version numbers that did not follow the definition of the Fixed type.) Version16Dot16 is a packed value: the upper 16 bits comprise a major version number, and the lower 16 bits, a minor version. Non-zero minor version numbers are represented using digits 0 to 9 in the highest-order nibbles of the lower 16 bits. For example, the version field of 'maxp' table version 0.5 is 0x00005000, and that of 'vhea' table version 1.1 is 0x00011000. This type is used only in the 'maxp', 'post' and 'vhea' tables, and will not be used for any other tables in the future.

The table directory uses a uint32 field, `sfntVersion`, with an enumeration of defined values, some of which represent four-character tags; see subclause 4.5 (Top-level OFF organization) for details.

Implementations reading tables shall include code to check version numbers so that, if and when the format and version number changes, older implementations will handle newer versions gracefully.

Minor version number changes always imply format changes that are compatible extensions. If an implementation understands a major version number, then it can safely proceed with reading the table. If the minor version is greater than the latest version recognized by the implementation, then the extension fields will be undetectable to the implementation.

For purposes of compatibility, version numbers that are represented using a single uint16 or uint32 value are treated like a minor version number, and any format changes are compatible extensions.

Some field values that were undefined or reserved in an earlier revision can be assigned meanings in a minor version change. Implementations should never make assumptions regarding reserved or unassigned values or bits in bit fields, and can ignore them if encountered. When writing font data, tools should always write zero for reserved fields or bits. Validators should warn of any non-zero values for fields or bits that are not defined for the given version against which data is being validated.

If the major version is not recognized, the implementation shall not read the table as it can make no assumptions regarding interpretation of the binary data. The implementation should treat the table as missing.

4.5 Top-level OFF organization

4.5.1 Table directory

A key characteristic of the OFF format is the TrueType sfnt "wrapper", which provides organization for a collection of tables in a general and extensible manner.

The OFF font starts with the table directory, which is a directory of the top-level tables in a font. If the font file contains only one font, the table directory will begin at byte 0 of the file. If the font file is a font collection (see 4.6), the beginning point of the table directory for each font is indicated in the TTCHHeader.

TableDirectory:

Type	Name	Description
uint32	sfntVersion	0x00010000 or 0x4F54544F ('OTTO') – see below.
uint16	numTables	Number of tables.
uint16	searchRange	Maximum power of 2 less than or equal to numTables, times 16 $((2^{**\text{floor}(\log_2(\text{numTables}))}) * 16)$, where ** is an exponentiation operator).
uint16	entrySelector	$\log_2$ of the maximum power of 2 less than or equal to numTables ($\log_2(\text{searchRange}/16)$, which is equal to $\text{floor}(\log_2(\text{numTables}))$).
uint16	rangeShift	numTables times 16, minus searchRange $(\text{numTables} * 16) - \text{searchRange}$.
TableRecord	tableRecords[numTables]	Table records array—one for each top-level table in the font.

NOTE In the preceding table, the symbol ** is an exponentiation operator, as used in several programming languages.

OFF fonts that contain TrueType outlines should use the value of 0x00010000 for sfntVersion. OFF fonts containing CFF data (version 1 or 2) should use 0x4F54544F ('OTTO', when re-interpreted as a Tag) for sfntVersion.

Apple's TrueType Reference Manual^[7] allows for 'true' and 'typ1' for sfntVersion. These version tags shall not be used for OFF fonts.

The table directory format allows for a large number of tables. To assist in quick binary searches, the searchRange, entrySelector and rangeShift fields are included as parameters that can be used in configuring search algorithms. Binary search is optimal when the number of entries as a power of two. The searchRange field provides the largest number of items that can be searched with that constraint (maximum power of two). The rangeShift field provides the remaining number of items that would also need to be searched. The entrySelector field indicates the maximum number of levels into the binary tree will need to be entered. Values are multiplied by 16, which is the size of each TableRecord.

In early implementations on devices with limited hardware capabilities, optimizations provided by the `searchRange`, `entrySelector` and `rangeShift` fields were of high importance. They have less importance on modern devices, but can still be used in some implementations. However, incorrect values could potentially be used as an attack vector against some implementations. Since these values can be derived from the `numTables` field when the file is parsed, it is strongly recommended that parsing implementations not rely on the `searchRange`, `entrySelector` and `rangeShift` fields in the font but derive them independently from `numTables`. Font files, however, should continue to provide valid values for these fields to maintain compatibility with all existing implementations.

4.5.2 Table records

The table directory ends with the `tableRecords` array. The `TableRecord` format is as follows.

TableRecord:

Type	Name	Description
Tag	<code>tableTag</code>	Table identifier.
uint32	<code>checksum</code>	Checksum for this table.
Offset32	<code>offset</code>	Offset from beginning of OFF font file.
uint32	<code>length</code>	Length of this table.

Table tags are the names given to tables in the OFF font file. The `tableRecords` array makes it possible for a given font to contain only those tables it actually needs. As a result, there is no standard value for `numTables`. The records in the array shall be sorted in ascending order by tag.

For requirements of Tag values, see 4.3 (Data types). Several tags and their associated table formats are defined in this specification. For table tags defined in this specification, a font resource should have at most one table record using a given tag. If a font resource does contain more than one table of a given type, behaviour is unpredictable: apps or platforms can select one of the tables arbitrarily, or can reject the font as invalid.

Platform vendors may define additional tables and associated tags to provide platform-specific functionality. See, for example, Apple's TrueType Reference Manual ^[7] which defines various tables with associated tags not defined in this document. Some font development tools may also define special tables. Fonts that contain such additional tables can still qualify as OFF fonts if they satisfy requirements of this document. For custom tables defined outside this document, an external specification of such a table may permit multiple tables of that type within a single font resource.

NOTE 1 The OpenType specification recommends that vendors notify Microsoft when they define custom tags to ensure forward compatibility in the event of future expansion of OpenType.

NOTE 2 Apple's TrueType specification ^[7] stipulates that tag names consisting of all lower case letters are reserved for Apple's use.

All tables shall begin on four-byte boundaries, and any remaining space between tables shall be padded with zeros. The length of each table should be recorded in the table record with the actual length of data, not the padded length.

NOTE 3 The requirement for four-byte alignment applies to top-level tables only and does not extend to sub-table offsets, records, or fields within tables or records.

Some tables have an internal structure with subtables located at specified offsets, and as a result, it is possible to construct a font with data for different tables interleaved. In general, tables should be arranged contiguously without overlapping the ranges of distinct tables. In any case, however, table lengths measure a contiguous range of bytes that encompasses all of the data for a table. This applies to any subtables as well as to top-level tables.

4.5.3 Calculating checksums

Table checksums are the unsigned sum of the uint32 units of a given table. In C, the following function can be used to determine a checksum:

```
uint32
CalcTableChecksum(uint32 *Table, uint32 Length)
{
    uint32 Sum = 0L;
    uint32 *EndPtr = Table+((Length+3) & ~3) / sizeof(uint32);

    while (Table < EndPtr)
        Sum += *Table++;
    return Sum;
}
```

This function assumes that the length of any table is a multiple of four bytes, or that tables are padded with zero to four-byte aligned offsets. Actual table lengths recorded in the table directory should not include padding, however. To accommodate data with a length that is not a multiple of four, the above algorithm should be modified to treat the data as though it contains zero padding to a length that is a multiple of four.

A checksum value of 0 (zero) indicates that the checksum has not been calculated for this table. A 0 value should be treated as a valid checksum for any conforming pair of offset and length values.

The 'head' table is a special case in checksum calculations, as it includes a checksumAdjustment field that is calculated and written after the table's checksum is calculated and written into the table directory entry, necessarily invalidating that checksum value.

When generating font data, to calculate and write the 'head' table checksum and checksumAdjustment field, do the following:

- Set the checksumAdjustment field to 0.
- Calculate the checksum for all tables including the 'head' table and enter the value for each table into the corresponding record in the table directory.
- Calculate the checksum for the entire font.
- Subtract that value from 0xB1B0AFBA.
- Store the result in the 'head' table checksumAdjustment field.

An application attempting to verify that the 'head' table has not changed should calculate the checksum for that table assuming that the checksumAdjustment value is zero, rather than the actual value in the font, before comparing the result with the 'head' table record in the table directory.

Within a font collection file (see 4.6), table checksums shall reflect the tables as they are in the collection file. The checksumAdjustment field in the 'head' table is not used for collection files and may be set to zero.

4.6 Font collections

4.6.1 The Font Collection overview

A Font Collection (either TTC or OTC, formerly known as TrueType Collection) is a means of delivering multiple OFF font resources in a single file structure. The format for font collections allows font tables that are identical between two or more fonts to be shared. Font collections containing outline glyph data (TrueType, CFF, CFF2 or SVG) are most useful when the fonts to be delivered together share many glyphs in common. By allowing multiple fonts to share glyph sets and other common font tables, font collections can result in a significant saving of file space.

For example, a group of Japanese fonts can each have their own designs for the kana glyphs, but share identical designs for the kanji. With ordinary OFF font files, the only way to include the common kanji glyphs is to copy their glyph data into each font. Since the kanji represent much more data than the kana, this results in a great deal of wasteful duplication of glyph data. Font collections were defined to solve this problem.

NOTE 1 Even though the original definition of Font Collection (as part of the TrueType specification) was intended to be used with fonts containing TrueType outlines, and this constraint was maintained in earlier editions of OFF, this is no longer a constraint in OFF. Font Collection files can contain various types of outlines (or a mix of them), regardless of whether or not fonts have layout tables present. For backward compatibility and simplicity, the description of the font collection file structure is using the term "TrueType Collection", or TTC, though it is understood that it is used to identify a generic font collection structure containing any type of outline tables.

NOTE 2 A variable font is functionally equivalent to multiple non-variable fonts. Variable fonts do not need to be contained within a collection file. A collection file can include one or even multiple variable fonts, however, and can even combine variable and non-variable fonts.

4.6.2 The Font Collection file structure

A font collection file consists of a single TTC header table, one or more table directories (each corresponding to a different font resource), and a number of OFF tables. The TTC header shall be located at the beginning of the TTC file.

The TTC file shall contain a complete table directory for each font resource. The same TableDirectory format is used for each font resource in a collection file as is used in a non-collection file. The table offsets in all table directories within a TTC file are measured from the beginning of the TTC file.

Each OFF table in a TTC file is referenced through the table directory of each font which uses that table. Some of the OFF tables could appear multiple times, once for each font included in the TTC; while other tables could be shared by multiple fonts in the TTC.

As an example, consider a TTC file which combines two Japanese fonts (Font1 and Font2). The fonts have different kana designs (Kana1 and Kana2) but use the same design for kanji. The TTC file contains a single 'glyf' table which includes both designs of kana together with the kanji; both fonts' table directories point to this 'glyf' table. But each font's table directory points to a different 'cmap' table, which identifies the glyph set to use. Font1's 'cmap' table points to the Kana1 region of the 'loca' and 'glyf' tables for kana glyphs, and to the kanji region for the kanji. Font2's 'cmap' table points to the Kana2 region of the 'loca' and 'glyf' tables for kana glyphs, and to the same kanji region for the kanji.

The tables that should have a unique copy per font are those that are used by the system in identifying the font and its character mapping, including 'cmap', 'name', and 'OS/2'. The tables that should be shared by fonts in the TTC are those that define glyph and instruction data or use glyph indices to access data: 'glyf', 'loca', 'hmtx', 'hdmx', 'LTSH', 'cvt', 'fpgm', 'prep', 'EBLC', 'EBDT', 'EBSC', 'maxp', and so on. In practice, any tables which have identical data for two or more fonts may be shared.

Each font in the TTC may contain its own DMAP table. Such a font can support characters and GlyphIDs also defined by another font; in the case of overlapping coverage the order of precedence is first DMAP, then 'cmap'.

NOTE When building a collection file from separate font files, close attention needs to be paid to the issue of glyph renumbering in a font and the side effects that can result in the 'cmap' table and elsewhere. The fonts to be merged also need to have compatible TrueType instructions; that is, their preprograms, function definitions, and control values cannot conflict.

Collection files containing TrueType glyph outlines should use the filename suffix .TTC. Collection files containing CFF or CFF2 outlines should use the file extension .OTC.

4.6.3 TTC header

There are four versions of the TTC header: Version 1.0 has been used for TTC files without digital signatures. Version 2.0 can be used for TTC files with *or* without digital signatures — if there's no signature, then the last three fields of the version 2.0 header are left null. Versions 1.1 and 2.1 mirror versions 1 and 2 but are used for supporting collections containing fonts with 24-bit glyph identifiers. Software that does not support versions 1.1 and 2.1 of the TTC header will continue to function using a version 1 or 2 header, if present, and will use the 16-bit tables pointed to from that header. Newer software will make use of the 24-bit tables. Software that supports *tableDirectoryOffsets2* shall use it instead of *tableDirectoryOffsets* when both are present.

Tables pointed to in collections using header versions 1.1 or 2.1 can overlap, so that, for example, much of the GPOS table could be shared between 16-bit and 24-bit fonts in the collection.

If a digital signature is used, the DSIG table for the file shall be located at the end of the TTC file, following any other font tables. Signatures in a TTC file are expected to be Format 1 signatures.

The purpose of the TTC header table is to locate the different table directories within a TTC file. The TTC header is located at the beginning of the TTC file (offset = 0). It consists of an identification tag, a version number, a count of the number of OFF fonts in the file, and an array of offsets to each table directory.

TTCHeader Version 1.0:

Type	Name	Description
Tag	ttcTag	Font Collection ID string: 'ttcf' (used for fonts with CFF or CFF2 outlines, as well as TrueType outlines)
uint16	majorVersion	Major version of the TTCHeader, = 1.
uint16	minorVersion	Minor version of the TTCHeader, = 0.
uint32	numFonts	Number of fonts in TTC

Type	Name	Description
Offset32	tableDirectoryOffsets [numFonts]	Array of offsets to the TableDirectory for each font from the beginning of the file

TTCHeader Version 1.1:

Type	Name	Description
Tag	ttcTag	Font Collection ID string: 'ttcf' (used for fonts with CFF or CFF2 outlines, as well as TrueType outlines)
uint16	majorVersion	Major version of the TTCHeader, = 1.
uint16	minorVersion	Minor version of the TTCHeader, = 1.
uint32	numFonts	Number of fonts in TTC
Offset32	tableDirectoryOffsets [numFonts]	Array of offsets to the TableDirectory for each font from the beginning of the file
uint32	numFonts2	Number of fonts in TTC that are using 24-bit glyph IDs
Offset32	tableDirectoryOffsets2 [numFonts2]	Array of offsets to the TableDirectory for each font using 24-bit glyph IDs, from the beginning of the file

TTCHeader Version 2.0:

Type	Name	Description
Tag	ttcTag	Font Collection ID string: 'ttcf'
uint16	majorVersion	Major version of the TTCHeader, = 2.
uint16	minorVersion	Minor version of the TTCHeader, = 0.
uint32	numFonts	Number of fonts in TTC
Offset32	tableDirectoryOffsets [numFonts]	Array of offsets to the TableDirectory for each font from the beginning of the file
uint32	dsigTag	Tag indicating that a DSIG table exists, 0x44534947 ('DSIG') (null if no signature)
uint32	dsigLength	The length (in bytes) of the DSIG table (null if no signature)
uint32	dsigOffset	The offset (in bytes) of the DSIG table from the beginning of the TTC file (null if no signature)

TTCHeader Version 2.1:

Type	Name	Description
Tag	ttcTag	Font Collection ID string: 'ttcf'
uint16	majorVersion	Major version of the TTCHeader, = 2.
uint16	minorVersion	Minor version of the TTCHeader, = 1.
uint32	numFonts	Number of fonts in TTC
Offset32	tableDirectoryOffsets [numFonts]	Array of offsets to the TableDirectory for each font from the beginning of the file
uint32	dsigTag	Tag indicating that a DSIG table exists, 0x44534947 ('DSIG') (null if no signature)
uint32	dsigLength	The length (in bytes) of the DSIG table (null if no signature)
uint32	dsigOffset	The offset (in bytes) of the DSIG table from the beginning of the TTC file (null if no signature)
uint32	numFonts2	Number of fonts in TTC that are using 24-bit glyph IDs
Offset32	tableDirectoryOffsets2 [numFonts2]	Array of offsets to the TableDirectory for each font using 24-bit glyph IDs, from the beginning of the file

5 Open font tables

5.1 Required common tables

5.1.1 List of required tables

Whether TrueType or CFF outlines are used in an OFF font, the following tables are required for the font to function correctly:

Tag	Name
cmap	Character to glyph mapping
head	Font header
hhea	Horizontal header
hmtx	Horizontal metrics
maxp	Maximum profile

Tag	Name
name	Naming table
OS/2	OS/2 and Windows specific metrics
post	PostScript information

5.1.2 cmap—Character to glyph index mapping table

5.1.2.1 Table overview

This table defines the mapping of character codes to a default glyph index. Different subtables may be defined that each contain mappings for different character encoding schemes. The table header indicates the character encodings for which subtables are present.

Regardless of the encoding scheme, character codes that do not correspond to any glyph in the font should be mapped to glyph index 0. The glyph at this location shall be a special glyph representing a missing character, commonly known as .notdef.

Each subtable is in one of seven possible formats and begins with a format field indicating the format used. The first four formats — formats 0, 2, 4 and 6 — were originally defined prior to Unicode 2.0. These formats allow for 8-bit single-byte, 8-bit multi-byte, and 16-bit encodings. With the introduction of supplementary planes in Unicode 2.0, the Unicode addressable code space extends beyond 16 bits. To accommodate this, three additional formats were added — formats 8, 10 and 12 — that allow for 32-bit encoding schemes.

Other enhancements in Unicode led to the addition of other subtable formats. Subtable format 13 allows for an efficient mapping of many characters to a single glyph; this is useful for “last-resort” fonts that provide fallback rendering for all possible Unicode characters with a distinct fallback glyph for different Unicode ranges. Subtable format 14 provides a unified mechanism for supporting Unicode variation sequences. Subtable format 15 was added to allow more than 65535 glyphs in a font; it is similar to format 14 but uses 24-bit glyph indices.

Of the ten available formats, not all are commonly used today. Formats 4 or 12 are appropriate for most new fonts, depending on the Unicode character repertoire supported. Format 14 is used in many applications for support of Unicode variation sequences. Format 15 is needed for larger fonts. Some platforms also make use for format 13 for a last-resort fallback font. Other subtable formats are not recommended for use in new fonts. Application developers, however, should anticipate that any of the formats can be used in fonts.

NOTE The 'cmap' table version number remains at 0x0000 for fonts that make use of the newer subtable formats.

An optional DMAP table, if present, shall take priority over 'cmap'—that is, the character-to-glyph lookup shall first look in the DMAP table (including any variation-selectors). If a match is not found, 'cmap' is to be consulted. The DMAP table is identical in structure to the 'cmap' table.

5.1.2.2 'cmap' Header

The Character to Glyph Index Mapping Table is organized as follows.

'cmap' Header:

Type	Name	Description
uint16	version	Table version number (0)
uint16	numTables	Number of encoding tables that follow
EncodingRecord	encodingRecords[numTables]	

5.1.2.3 Encoding records and encodings

The array of encoding records specifies particular encodings and the offset to the subtable for each encoding.

EncodingRecord:

Type	Name	Description
uint16	platformID	Platform ID.
uint16	encodingID	Platform-specific encoding ID.
Offset32	subtableOffset	Byte offset from beginning of table to the subtable for this encoding.

The platform ID and platform-specific encoding ID in the encoding record are used to specify a particular character encoding. In the case of the Macintosh platform, a language field within the mapping subtable is also used for this purpose.

The encoding record entries in the 'cmap' header shall be sorted first by platform ID, then by platform-specific encoding ID, and then by the language field in the corresponding subtable. Each platform ID, platform-specific encoding ID, and subtable language combination may appear only once in the 'cmap' table.

Apart from a format 14 subtable, all other subtables are exclusive: applications should select and use one and ignore the others. If a Unicode subtable is used (platform 0, or platform 3 / encoding 1 or 10), then a format 14 subtable using platform 0/encoding 5 can also be supplemented for mapping Unicode variation sequences.

If a font includes Unicode subtables for both 16-bit encoding (typically, format 4) and also 32-bit encoding (formats 10 or 12), then the characters supported by the subtable for 32-bit encoding should be a superset of the characters supported by the subtable for 16-bit encoding, and the 32-bit encoding should be used by applications.

Fonts should not include 16-bit Unicode subtables using both format 4 and format 6; format 4 should be used. Similarly, fonts should not include 32-bit Unicode subtables using both format 10 and format 12; format 12 should be used.

If a font includes encoding records for Unicode subtables of the same format but with different platform IDs, an application may choose which to select, but should make this selection consistently each time the font is used.

5.1.2.4 Platform IDs

The following platform IDs are defined:

Platform ID	Platform name	Platform-specific encoding IDs
0	Unicode	Various
1	Macintosh	Script manager code
2	<i>ISO [deprecated]</i>	<i>ISO encoding [deprecated]</i>
3	Windows	Windows encoding
4	Custom	Custom

Platform ID values 240 through 255 are reserved for user-defined platforms, and shall never be assigned to a registered platform.

Unicode platform (platform ID = 0)

The following encoding IDs are defined for use with the Unicode platform:

Encoding ID	Description
0	Unicode 1.0 semantics— <i>deprecated</i>
1	Unicode 1.1 semantics— <i>deprecated</i>
2	ISO/IEC 10646 semantics— <i>deprecated</i>
3	Unicode 2.0 and onwards semantics, Unicode BMP only
4	Unicode 2.0 and onwards semantics, Unicode full repertoire
5	Unicode variation sequences—for use with subtable format 14
6	Unicode full repertoire—for use with subtable format 13

Use of encoding IDs 0, 1 or 2 is deprecated.

Encoding ID 3 should be used in conjunction with 'cmap' subtable formats 4 or 6. Encoding ID 4 should be used in conjunction with subtable formats 10 or 12.

Unicode variation sequences supported by the font should be specified in the 'cmap' table using a format 14 subtable. A format 14 subtable shall only be used under platform ID 0 and encoding ID 5; and encoding ID 5 should only be used with a format 14 subtable.

Encoding ID 6 should only be used in conjunction with 'cmap' subtable format 13; and subtable format 13 should only be used under platform ID 0 and encoding ID 6.

Macintosh platform (platform ID = 1)

Older Macintosh versions required fonts to have a 'cmap' subtable for platform ID 1. For current Apple platforms, use of platform ID 1 is discouraged. See 5.1.7.3 for details regarding encoding IDs defined for the Macintosh platform.

ISO platform (platform ID = 2)

Use of this platform ID is deprecated.

The following encoding IDs are defined for use with ISO platform:

Code	ISO encoding
0	7-bit ASCII
1	ISO 10646
2	ISO 8859-1

Windows platform (platform ID = 3)

The Windows platform supports several encodings. When creating fonts for Windows, Unicode 'cmap' subtables should always be used—platform ID 3 with encoding ID 1 or encoding ID 10. See below for additional details.

The following encoding IDs are supported on the Windows platform:

Platform ID	Encoding ID	Description
3	0	Symbol
3	1	Unicode BMP
3	2	ShiftJIS
3	3	PRC
3	4	Big5
3	5	Wansung
3	6	Johab
3	7	Reserved
3	8	Reserved
3	9	Reserved
3	10	Unicode full repertoire

Fonts that support only Unicode BMP characters (U+0000 to U+FFFF) on the Windows platform shall use encoding 1 with a format 4 subtable. This encoding shall not be used to support Unicode supplementary-plane characters.

Fonts that support Unicode supplementary-plane characters (U+10000 to U+10FFFF) on the Windows platform shall use encoding 10 with a format 12 subtable.

The symbol encoding was created to support fonts with arbitrary ornaments or symbols not supported in Unicode or other standard encodings. A format 4 subtable would be used, typically with up to 224 graphic characters assigned at code positions beginning with 0xF020. This corresponds to a sub-range within the Unicode Private-Use Area (PUA), though this is not a Unicode encoding. In legacy usage, some applications would represent the symbol characters in text using a single-byte encoding, and then map 0x20 to the OS/2.usFirstCharIndex value in the font. In new fonts, symbols or characters not in Unicode should be encoded using PUA code points in a Unicode 'cmap' subtable.

See the recommendations in 8.7 for additional information.

Custom platform (platform ID = 4) and OTF Windows NT compatibility mapping

Platform ID 4 is a legacy platform that was created to provide compatibility of older applications with OFF fonts that had been adapted from older Type 1 fonts. This platform is not commonly used today, and should not be used in new fonts.

ID	Custom encoding
0-255	OTF Windows NT compatibility mapping

This 'cmap' platform provides a compatibility mechanism for non-Unicode applications that use the font as if it were Windows ANSI encoded. Non-Windows ANSI Type 1 fonts, such as Cyrillic and Central European fonts that Adobe shipped in the past, had "0" (Windows ANSI) recorded in the CharSet field of the .PFM file; Adobe Type Manager for Windows 9x ignored the CharSet altogether. Adobe provides this compatibility 'cmap' encoding in every OFF font converted from a Type1 font in which the Encoding is not StandardEncoding.

When platform ID 4 is used, the encoding ID shall be set to the Windows charset value (in the range 0 to 255, inclusive) present in the .PFM file of the original Type 1 font.

If a platform ID 4, encoding ID 0 – 255 'cmap' encoding is present in an OFF font with CFF outlines, then the OTF font driver in Windows NT will: (a) superimpose the glyphs encoded at character codes 0-255 in the encoding on the corresponding Windows ANSI (code page 1252) Unicode values in the Unicode encoding it reports to the system; (b) add Windows ANSI (CharSet 0) to the list of CharSets supported by the font; and (c) consider the value of the encoding ID to be a Windows CharSet value and add it to the list of CharSets supported by the font. Note that the 'cmap' subtable needs to use Format 0 or 6 for its subtable, and the encoding needs to be identical to the CFF's encoding.

5.1.2.5 'cmap' subtable formats

5.1.2.5.1 Use of the language field in 'cmap' subtables

All 'cmap' subtable formats include a language field. The language field shall be set to zero for all cmap subtables whose platform IDs are other than Macintosh (platform ID 1). For cmap subtables whose platform IDs are Macintosh, set this field to the Macintosh language ID of the cmap subtable plus one, or to zero if the cmap subtable is not language-specific. For example, a Mac OS Turkish cmap subtable could set this field to 18, since the Macintosh language ID for Turkish is 17. A Mac OS Roman 'cmap' subtable shall set this field to 0, since Mac OS Roman is not a language-specific encoding.

5.1.2.5.2 Format 0: Byte encoding table

Format 0 was the standard mapping subtable used on older Macintosh platforms (see Reference [7] for details) but is not required on newer Apple platforms.

'cmap' Subtable Format 0:

Type	Name	Description
uint16	format	Format number is set to 0.
uint16	length	This is the length in bytes of the subtable.
uint16	language	For requirements on use of the language field, see subclause 5.1.2.5.1.
uint8	glyphIdArray[256]	An array that maps character codes to glyph index values.

This is a simple 1 to 1 mapping of character codes to glyph indices. The glyph set is limited to 256. If this format is used to index into a larger glyph set, only the first 256 glyphs will be accessible.

5.1.2.5.3 Format 2: High byte mapping through table

This subtable format was created for “double-byte” encodings following national character code standards used for Japanese, Chinese, and Korean characters. These code standards use a mixed 8-/16-bit encoding. This format is not commonly used today.

In these mixed 8-/16-bit encodings, certain byte values signal the first byte of a 2-byte character. (These byte values are also valid as the second byte of a 2-byte character.) In addition, even for the 2-byte characters, the mapping of character codes to glyph index values depends heavily on the first byte. Consequently, the table begins with an array that maps the first byte to a SubHeader record. For 2-byte character codes, the SubHeader is used to map the second byte's value into a sub-range of a glyph index array—a *sub-array*—, as described below. When processing mixed 8-/16-bit text, SubHeader 0 is special: it is used for single-byte character codes. When SubHeader 0 is used, a second byte is not needed; the single byte value is mapped through the specified sub-array.

'cmap' Subtable Format 2:

Type	Name	Description
uint16	format	Format number is set to 2.
uint16	length	This is the length in bytes of the subtable.
uint16	language	For requirements on use of the language field, see subclause 5.1.2.5.1.
uint16	subHeaderKeys[256]	Array that maps high bytes into the subHeaders array: value is subHeader index $\times$ 8.
SubHeader	subHeaders[]	Variable-length array of SubHeader records.
uint16	glyphIdArray[]	Variable-length array containing sub-arrays used for mapping the low byte of 2-byte characters.

The SubHeader record is structured as follows:

SubHeader Record:

Type	Name	Description
uint16	firstCode	First valid low byte for this SubHeader.
uint16	entryCount	Number of valid low bytes for this SubHeader.
int16	idDelta	See text below.
uint16	idRangeOffset	See text below.

The firstCode and entryCount values specify a subrange that begins at firstCode and has a length equal to the value of entryCount. This subrange stays within the 0-255 range of the byte being mapped. Bytes outside of this subrange are mapped to glyph index 0 (missing glyph). The offset of the byte within this subrange is then used as index into a corresponding sub-array of glyphIdArray. This sub-array is also of length entryCount. The value of the idRangeOffset is the number of bytes past the actual location of the idRangeOffset word where the glyphIdArray element corresponding to firstCode appears.

Finally, if the value obtained from the sub-array is not 0 (which indicates the missing glyph), you should add idDelta to it in order to get the glyphIndex. The value idDelta permits the same sub-array to be used for several different subheaders. The idDelta arithmetic is modulo 65536. If the result after adding idDelta to the value from the sub-array is less than zero, add 65536 to obtain a valid glyph ID.

5.1.2.5.4 Format 4: Segment mapping to delta values

This is the standard character-to-glyph-index mapping subtable for fonts that support only Unicode Basic Multilingual Plane characters (U+0000 to U+FFFF).

To support Unicode supplementary-plane characters, format 12 should be used.

This format is used when the character codes for the characters represented by a font fall into several contiguous ranges, possibly with holes in some or all of the ranges (that is, some of the codes in a range might not have a representation in the font). The format-dependent data is divided into three parts, which shall occur in the following order:

- A four-word header gives parameters for an optimized search of the segment list.
- Four parallel arrays describe the segments (one segment for each contiguous range of codes).
- A variable-length array of glyph IDs (unsigned words).

'cmap' Subtable Format 4:

Type	Name	Description
uint16	format	Format number is set to 4.
uint16	length	This is the length in bytes of the subtable.
uint16	language	For requirements on use of the language field, see subclause 5.1.2.5.1.
uint16	segCountX2	$2 \times \text{segCount}$.
uint16	searchRange	Maximum power of 2 less than or equal to segCount, times 2 $((2^{\text{floor}(\log_2(\text{segCount}))}) * 2$, where “**” is an exponentiation operator)

Type	Name	Description
uint16	entrySelector	$\log_2$ of the maximum power of 2 less than or equal to $\text{segCount}(\log_2(\text{searchRange}/2))$, which is equal to $\text{floor}(\log_2(\text{segCount}))$
uint16	rangeShift	segCount times 2, minus searchRange $((\text{segCount} * 2) - \text{searchRange})$
uint16	endCode[segCount]	End characterCode for each segment, last=0xFFFF.
uint16	reservedPad	Set to 0.
uint16	startCode[segCount]	Start character code for each segment.
int16	idDelta[segCount]	Delta for all character codes in segment.
uint16	idRangeOffset[segCount]	Offsets into glyphIdArray or 0
uint16	glyphIdArray[]	Glyph index array (arbitrary length)

The number of segments is specified by `segCount`, which is not given directly in the header but is readily derived from `segCountX2`. All of the other header parameters are derived from it. The `searchRange` value is twice the largest power of 2 that is less than or equal to `segCount`. For example, if `segCount=39`, we have the following:

```
segCountX2    78
searchRange   64  (= 2 × (largest power of 2 ≤ 39))
entrySelector  5   (= log232)
rangeShift    14  (= 2 × 39 - 64)
```

To assist in quick binary searches, the `searchRange`, `entrySelector` and `rangeShift` fields are included as parameters that can be used in configuring search algorithms. In particular, binary search is optimal when the number of entries is a power of two. The `searchRange` field provides the largest number of items that can be searched with that constraint (maximum power of two). The `rangeShift` field provides the remaining number of items that would also need to be searched. The `entrySelector` field indicates the maximum number of levels into the binary tree will need to be entered.

In early implementations on devices with limited hardware capabilities, optimizations provided by the `searchRange`, `entrySelector` and `rangeShift` fields were of high importance. They have less importance on modern devices, but could still be used in some implementations. However, incorrect values could potentially be used as an attack vector against some implementations. Since these values can be derived from the `segCountX2` field when the file is parsed, it is strongly recommended that parsing implementations not rely on the `searchRange`, `entrySelector` and `rangeShift` fields in the font but derive them independently from `segCountX2`. Font files, however, should continue to provide valid values for these fields to maintain compatibility with all existing implementations.

Each segment is described by a `startCode` and `endCode`, along with an `idDelta` and an `idRangeOffset`, which are used for mapping the character codes in the segment. The segments are sorted in order of increasing `endCode` values, and the segment values are specified in four parallel arrays. You search for the first `endCode` that is greater than or equal to the character code you want to map. If the corresponding `startCode` is less than or equal to the character code, then you use the corresponding

idDelta and idRangeOffset to map the character code to a glyph index (otherwise, the missingGlyph is returned). For the search to terminate, the final startCode and endCode values shall be 0xFFFF. This segment need not contain any valid mappings. (It can just map the single character code 0xFFFF to missingGlyph). However, the segment shall be present.

If the idRangeOffset value for the segment is not 0, the mapping of character codes relies on glyphIdArray. The character code offset from startCode is added to the idRangeOffset value. This sum is used as an offset from the current location within idRangeOffset itself to index out the correct glyphIdArray value. This obscure indexing trick works because glyphIdArray immediately follows idRangeOffset in the font file. The C expression that yields the glyph index is:

```
glyphId = *(idRangeOffset[i]/2
           + (c - startCode[i])
           + &idRangeOffset[i])
```

The value *c* is the character code in question, and *i* is the segment index in which *c* appears. If the value obtained from the indexing operation is not 0 (which indicates missingGlyph), idDelta[i] is added to it to get the glyph index. The idDelta arithmetic is modulo 65536.

If the idRangeOffset is 0, the idDelta value is added directly to the character code offset (i.e. idDelta[i] + *c*) to get the corresponding glyph index. Again, the idDelta arithmetic is modulo 65536. If the result after adding idDelta[i] + *c* is less than zero, add 65536 to obtain a valid glyph ID.

As an example, the variant part of the table to map characters 10-20, 30-90, and 153-480 onto a contiguous range of glyph indices can look like this:

segCountX2:	8
searchRange:	8
entrySelector:	2
rangeShift:	0
endCode:	20 90 480 0xffff
reservedPad:	0
startCode:	10 30 153 0xffff
idDelta:	-9 -18 -80 1
idRangeOffset:	0 0 0 0

This table yields the following mappings:

```
10 ⇒ 10 - 9 = 1
20 ⇒ 20 - 9 = 11
30 ⇒ 30 - 18 = 12
90 ⇒ 90 - 18 = 72
153 ⇒ 153 - 80 = 73
480 ⇒ 480 - 80 = 400
0xffff ⇒ 0
```

Note that the delta values could be reworked so as to reorder the segments.

5.1.2.5.5 Format 6: Trimmed table mapping

Format 6 was designed to map 16-bit characters to glyph indexes when the character codes for a font fall into a single contiguous range.

'cmap' Subtable Format 6:

Type	Name	Description
uint16	format	Format number is set to 6.
uint16	length	This is the length in bytes of the subtable.
uint16	language	For requirements on use of the language field, see subclause 5.1.2.5.1.
uint16	firstCode	First character code of subrange.
uint16	entryCount	Number of character codes in subrange.
uint16	glyphIdArray [entryCount]	Array of glyph index values for character codes in the range.

The firstCode and entryCount values specify a subrange (beginning at firstCode, length = entryCount) within the range of possible character codes. Codes outside of this subrange are mapped to glyph index 0. The offset of the code (from the first code) within this subrange is used as index to the glyphIdArray, which provides the glyph index value.

5.1.2.5.6 Format 8: mixed 16-bit and 32-bit coverage

Subtable format 8 was designed to support Unicode supplementary-plane characters in UTF-16 encoding, though it is not commonly used. Format 8 is similar to format 2 in that it provides for mixed-length character codes. Instead of allowing for 8- and 16-bit character codes, however, it allows for 16- and 32-bit character codes.

If a font contains Unicode supplementary-plane characters (U+10000 to U+10FFFF), then it's likely that it will also include Unicode BMP characters (U+0000 to U+FFFF) as well. Hence, there is a need to map a mixture of 16-bit and 32-bit character codes. A simplifying assumption is made: namely, that there are no 32-bit character codes which share the same first 16 bits as any 16-bit character code. (Since the Unicode code space extends only to U+10FFFF, a potential conflict exists only for characters U+0000 to U+0010, which are non-printing control characters.) This means that the determination as to whether a particular 16-bit value is a standalone character code or the start of a 32-bit character code can be made by looking at the 16-bit value directly, with no further information required.

'cmap' Subtable Format 8:

Type	Name	Description
uint16	format	Subtable format; set to 8.
uint16	reserved	Reserved; set to 0
uint32	length	Byte length of this subtable (including the header)

Type	Name	Description
uint32	language	For requirements on use of the language field, see subclause 5.1.2.5.1.
uint8	is32[8192]	Tightly packed array of bits (8K bytes total) indicating whether the particular 16-bit (index) value is the start of a 32-bit character code
uint32	numGroups	Number of groupings which follow
SequentialMapGroup	groups[numGroups]	Array of SequentialMapGroup records.

Each sequential map group record specifies a character range and the starting glyph ID mapped from the first character. Glyph IDs for subsequent characters follow in sequence.

SequentialMapGroup Record:

Type	Name	Description
uint32	startCharCode	First character code in this group; note that if this group is for one or more 16-bit character codes (which is determined from the is32 array), this 32-bit value will have the high 16-bits set to zero
uint32	endCharCode	Last character code in this group; same condition as listed above for the startCharCode
uint32	startGlyphID	Glyph index corresponding to the starting character code

A few notes here. The endCharCode is used, rather than a count, because comparisons for group matching are usually done on an existing character code, and having the endCharCode be there explicitly saves the necessity of an addition per group. Groups shall be sorted by increasing startCharCode. A group's endCharCode shall be less than the startCharCode of the following group, if any.

To determine if a particular word (cp) is the first half of 32-bit code points, one can use an expression such as $(\text{is32}[\text{cp} / 8] \& (1 << (7 - (\text{cp} \% 8))))$. If this is non-zero, then the word is the first half of a 32-bit code point.

0 is not a special value for the high word of a 32-bit code point. A font shall not have both a glyph for the code point 0x0000 and glyphs for code points with a high word of 0x0000.

The presence of the packed array of bits indicating whether a particular 16-bit value is the start of a 32-bit character code is useful even when the font contains no glyphs for a particular 16-bit start value. This is because the system software often needs to know how many bytes ahead the next character begins, even if the current character maps to the missing glyph. By including this information explicitly in this table, no "secret" knowledge needs to be encoded into the OS.

Although this format was created to support Unicode supplementary-plane characters, it is not widely supported or used. Also, no character encoding other than Unicode uses mixed 16-/32-bit characters. The use of this format is discouraged.

5.1.2.5.7 Format 10: Trimmed array

Subtable format 10 was designed to support Unicode supplementary-plane characters, though it is not commonly used. Format 10 is similar to format 6 in that it defines a trimmed array for a tight range of character codes. It differs, however, in that it uses 32-bit character codes.

'cmap' Subtable Format 10:

Type	Name	Description
uint16	format	Subtable format; set to 10.
uint16	reserved	Reserved; set to 0
uint32	length	Byte length of this subtable (including the header)
uint32	language	For requirements on use of the language field, see subclause 5.1.2.5.1.
uint32	startCharCode	First character code covered
uint32	numChars	Number of character codes covered
uint16	glyphIdArray []	Array of glyph indices for the character codes covered

This format is not widely used and is not supported on Windows platforms. It would be most suitable for fonts that support only a contiguous range of Unicode supplementary-plane characters, but such fonts are rare.

5.1.2.5.8 Format 12: Segmented coverage

This is the standard character-to-glyph-index mapping subtable for fonts supporting Unicode character repertoires that include supplementary-plane characters (U+10000 to U+10FFFF).

Fonts that include a format 12 subtable can also include a format 4 subtable for compatibility with older applications. This is not required, however. See the recommendations in 8.7 for additional information.

Format 12 is similar to format 4 in that it defines segments for sparse representation. It differs, however, in that it uses 32-bit character codes.

'cmap' Subtable Format 12:

Type	Name	Description
uint16	format	Subtable format; set to 12.
uint16	reserved	Reserved; set to 0
uint32	length	Byte length of this subtable (including the header)
uint32	language	For requirements on use of the language field, see subclause 5.1.2.5.1.

Type	Name	Description
uint32	numGroups	Number of groupings which follow
SequentialMapGroup	groups[numGroups]	Array of SequentialMapGroup records.

The sequential map group record is the same format as is used for the format 8 subtable. The qualifications regarding 16-bit character codes does not apply here, however, since characters codes are uniformly 32-bit.

SequentialMapGroup Record:

Type	Name	Description
uint32	startCharCode	First character code in this group
uint32	endCharCode	Last character code in this group
uint32	startGlyphID	Glyph index corresponding to the starting character code

Groups shall be sorted by increasing startCharCode. A group's endCharCode shall be less than the startCharCode of the following group, if any. The endCharCode is used, rather than a count, because comparisons for group matching are usually done on an existing character code, and having the endCharCode be there explicitly saves the necessity of an addition per group.

5.1.2.5.9 Format 13: Many-to-one range mappings

This subtable provides for situations in which the same glyph is used for hundreds or even thousands of consecutive characters spanning across multiple ranges of the code space. This subtable format can be useful for “last resort” fonts, although these fonts may use other suitable subtable formats as well. (For “last resort” fonts, see also the 'head' table flags in [subclause 5.1.3](#), bit 14).

NOTE Subtable formats 13 has the same data structure as format 12; it differs only in the interpretation of the startGlyphID/glyphID fields.

'cmap' Subtable Format 13:

Type	Name	Description
uint16	format	Subtable format; set to 13.
uint16	reserved	Reserved; set to 0
uint32	length	Byte length of this subtable (including the header)
uint32	language	For requirements on use of the language field, see subclause 5.1.2.5.1.
uint32	numGroups	Number of groupings which follow
ConstantMapGroup	groups[numGroups]	Array of ConstantMapGroup records.

The constant map group record has the same structure as the sequential map group record, with start and end character codes and a mapped glyph ID. However, the same glyph ID applies to all characters in the specified range rather than sequential glyph IDs.

ConstantMapGroup Record:

Type	Name	Description
uint32	startCharCode	First character code in this group
uint32	endCharCode	Last character code in this group
uint32	glyphID	Glyph index to be used for all the characters in the group's range.

5.1.2.5.10 Format 14: Unicode variation sequences

Subtable format 14 specifies the Unicode variation sequences (UVSes) supported by the font. A variation sequence, according to the Unicode Standard, comprises a base character followed by a variation selector. For example, <U+82A6, U+E0101>.

This subtable format shall only be used under platform ID 0 and encoding ID 5.

The subtable partitions the UVSes supported by the font into two categories: “default” and “non-default” UVSes. Given a UVS, if the glyph obtained by looking up the base character of that sequence in the Unicode 'cmap' subtable (i.e., the BMP subtable or BMP + supplementary-planes subtable) is the glyph to use for that sequence, then the sequence is a “default” UVS. Otherwise, it is a “non-default” UVS, and the glyph to use for that sequence is specified in the format 14 subtable itself.

The example at the end of this subclause shows how a font vendor can use format 14 for a JIS-2004-aware font.

'cmap' Subtable Format 14:

Type	Name	Description
uint16	format	Subtable format; set to 14.
uint32	length	Byte length of this subtable (including this header)
uint32	numVarSelectorRecords	Number of variation Selector Records
VariationSelector	varSelector[numVarSelectorRecords]	Array of VariationSelector records.

Each VariationSelector record specifies a variation selector character, and offsets to “default” and “non-default” tables used to map variation sequences using that variation selector.

VariationSelector Record:

Type	Name	Description
uint24	varSelector	Variation selector
Offset32	defaultUVSOffset	Offset from the start of the format 14 subtable to Default UVS table. May be 0.
Offset32	nonDefaultUVSOffset	Offset from the start of the format 14 subtable to Non-Default UVS table. May be 0.

The VariationSelector records are sorted in increasing order of varSelector. No two records may have the same varSelector value.

A VariationSelector record and the data its offsets point to specify those UVSes supported by the font for which the variation selector is the varSelector value of the record. The base characters of the UVSes are stored in the tables pointed to by the offsets. The UVSes are partitioned by whether they are default or non-default UVSes.

Glyph IDs to be used for non-default UVSes are specified in the Non-Default UVS table.

Default UVS table

A Default UVS table is simply a range-compressed list of Unicode scalar values, representing the base characters of the default UVSes which use the varSelector of the associated VariationSelector record.

Default UVS Table:

Type	Name	Description
uint32	numUnicodeValueRanges	Number of Unicode character ranges.
UnicodeRange	ranges[numUnicodeValueRanges]	Array of UnicodeRange records.

Each Unicode range record specifies a contiguous range of Unicode values.

UnicodeRange Record:

Type	Name	Description
uint24	startUnicodeValue	First value in this range
uint8	additionalCount	Number of additional values in this range

For example, the range U+4E4D – U+4E4F (3 values) will set startUnicodeValue to 0x004E4D and additionalCount to 2. A singleton range will set additionalCount to 0.

The sum (startUnicodeValue + additionalCount) shall not exceed 0xFFFFF.

The Unicode Value Ranges are sorted in increasing order of startUnicodeValue. The ranges shall not overlap; i.e., (startUnicodeValue + additionalCount) shall be less than the startUnicodeValue of the following range (if any).

All code points listed in the ranges array should have corresponding entries in a Unicode 'cmap' subtable. Applications could encounter fonts in which this is not the case, however.

Non-Default UVS table

A Non-Default UVS table is a list of pairs of Unicode scalar values and glyph IDs. The Unicode values represent the base characters of all non-default UVSes which use the varSelector of the associated VariationSelector record, and the glyph IDs specify the glyph IDs to use for the UVSes.

NonDefault UVS Table:

Type	Name	Description
uint32	numUVSMappings	Number of UVS Mappings that follow
UVSMapping	uvMappings[numUVSMappings]	Array of UVSMapping records.

Each UVSMapping record provides a glyph ID mapping for one base Unicode character, when that base character is used in a variation sequence with the current variation selector.

UVSMapping Record:

Type	Name	Description
uint24	unicodeValue	Base Unicode value of the UVS
uint16	glyphID	Glyph ID of the UVS

The UVS Mappings are sorted in increasing order of unicodeValue. No two mappings in this table may have the same unicodeValue values.

Typically, code points listed in the uvMappings array will have corresponding entries in a Unicode 'cmap' subtable. This is not required, however. For example, it might not be the case if a font is intended for use with content in which a given Unicode character only occurs in a variation sequence.

Example

Here is an example of how a format 14 'cmap' subtable can be used in a font that is aware of JIS-2004 variant glyphs. The CIDs (character IDs) in this example refer to those in the Adobe Character Collection "Adobe-Japan1", and can be assumed to be identical to the glyph IDs in the font in our example.

JIS-2004 changed the default glyph variants for some of its code points. For example:

JIS-90: U+82A6 ⇒ CID 1142

JIS-2004: U+82A6 ⇒ CID 7961

Both of these glyph variants are supported through the use of Unicode variation sequences, as the following examples from Unicode's UVS registry show:

U+82A6 U+E0100 ⇒ CID 1142

U+82A6 U+E0101 ⇒ CID 7961

If the font wants to support the JIS-2004 variants by default, it will:

- encode glyph ID 7961 at U+82A6 in the Unicode 'cmap' subtable,
- specify <U+82A6, U+E0101> in the UVS 'cmap' subtable's Default UVS table (varSelector will be 0x0E0101 and defaultUVSOffset will point to data containing a 0x0082A6 Unicode value)
- specify <U+82A6, U+E0100> $\Rightarrow$ glyph ID 1142 in the UVS 'cmap' subtable's Non-Default UVS table (varSelector will be 0x0E0100 and nonDefaultBaseUVSOffset will point to data containing a unicodeValue 0x0082A6 and glyphID 1142).

If, however, the font wants to support the JIS-90 variants by default, it will:

- encode glyph ID 1142 at U+82A6 in the Unicode 'cmap' subtable,
- specify <U+82A6, U+E0100> in the UVS 'cmap' subtable's Default UVS table
- specify <U+82A6, U+E0101> $\Rightarrow$ glyph ID 7961 in the UVS 'cmap' subtable's Non-Default UVS table

5.1.2.5.11 Format 15: Extended glyph repertoire UVS mapping

Subtable format 15 is the same as subtable format 14, except that it uses 24-bit index values in the UVSMapping Records.

UVSMapping24 Record:

Type	Name	Description
uint24	unicodeValue	Base Unicode value of the UVS
uint16	glyphID	Glyph ID of the UVS

The UVS Mappings are sorted in increasing order of unicodeValue. No two mappings in this table may have the same unicodeValue values. The *UVSMapping24 Record* format is intended for use with the 'GLYP' table.

5.1.3 head—Font header

This table gives global information about the font. The bounding box values should be computed using *only* glyphs that have contours. Glyphs with no contours should be ignored for the purposes of these calculations.

Type	Name	Description
uint16	majorVersion	Major version number of the font header table — set to 1.
uint16	minorVersion	Minor version number of the font header table — set to 0.
Fixed	fontRevision	Set by font manufacturer.

Type	Name	Description
uint32	checksumAdjustment	To compute: set it to 0, sum the entire font as uint32, then store 0xB1B0AFBA - sum. If the font is used as a component in a font collection file, the value of this field will be invalidated by changes to the file structure and font table directory, and shall be ignored.
uint32	magicNumber	Set to 0x5F0F3CF5.
uint16	flags	Bit 0: Baseline for font at y=0. Bit 1: Left sidebearing point at x=0 (relevant only for TrueType rasterizers) — see additional information below regarding variable fonts. Bit 2: Instructions can depend on point size. Bit 3: Force ppem to integer values for all internal scaler math; may use fractional ppem sizes if this bit is clear. It is strongly recommended that this be set in hinted fonts. Bit 4: Instructions can alter advance width (the advance widths might not scale linearly). Bit 5: This bit is not used in OFF, and should not be set in order to ensure compatible behavior on all platforms. If set, it can result in different behavior for vertical layout in some platforms. (See the 'head' table specification of "Apple's TrueType Reference Manual" [7] for details regarding behavior in Apple platforms.) Bits 6 – 10: These bits are not used in OFF and should always be cleared. (See the 'head' table specification of "Apple's TrueType Reference Manual" [7] for details regarding legacy use in Apple platforms.) Bit 11: Font data is “lossless” as a result of having been subjected to optimizing transformation and/or compression (such as font compression mechanisms defined by ISO/IEC 14496-18, MicroType®⁴

⁴ MicroType® is the trademark of a product supplied by Monotype Imaging Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

Type	Name	Description
		Express, WOFF2 [24], or similar) where the original font functionality and features are retained but the binary compatibility between input and output font files is not guaranteed. As a result of the applied transform, the 'DSIG' table can also be invalidated. Bit 12: Font converted (produce compatible metrics). Bit 13: Font optimized for ClearType®⁵. Fonts that rely on embedded bitmaps (EBDT) for rendering should not be considered optimized for ClearType, and therefore should keep this bit cleared. Bit 14: Last Resort font. If set, indicates that the glyphs encoded in the 'cmap' subtables are simply generic symbolic representations of code point ranges and do not truly represent support for those code points. If unset, indicates that the glyphs encoded in the cmap subtables represent proper support for those code points. Bit 15: Reserved, set to 0.
uint16	unitsPerEm	Set to a value from 16 to 16384. Any value in this range is valid. In fonts that have TrueType outlines, a power of 2 is recommended as this allows performance optimization in some rasterizers.
LONGDATETIME	created	Number of seconds since 12:00 midnight that started January 1 st , 1904, in GMT/UTC time zone.
LONGDATETIME	modified	Number of seconds since 12:00 midnight that started January 1 st , 1904, in GMT/UTC time zone.
int16	xMin	Minimum x coordinate across all glyph bounding boxes.
int16	yMin	Minimum y coordinate across all glyph bounding boxes.
int16	xMax	Maximum x coordinate across all glyph bounding boxes.

⁵ ClearType® is the trademark of a product supplied by Microsoft. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

Type	Name	Description
int16	yMax	Maximum y coordinate across all glyph bounding boxes.
uint16	macStyle	Bit 0: Bold (if set to 1); Bit 1: Italic (if set to 1) Bit 2: Underline (if set to 1) Bit 3: Outline (if set to 1) Bit 4: Shadow (if set to 1) Bit 5: Condensed (if set to 1) Bit 6: Extended (if set to 1) Bits 7 – 15: Reserved (set to 0).
uint16	lowestRecPPEM	Smallest readable size in pixels.
int16	fontDirectionHint ^a	Deprecated (Set to 2). 0: Fully mixed directional glyphs; 1: Only strongly left to right; 2: Like 1 but also contains neutrals; -1: Only strongly right to left; -2: Like -1 but also contains neutrals.
int16	indexToLocFormat	0 for short offsets (Offset16), 1 for long (Offset32).
int16	glyphDataFormat	0 for current format.

^a The fontDirectionHint field was intended for support of right-to-left scripts. Strong left to right, strong right to left, and neutral are properties of characters. A neutral character has no inherent directionality; it is not a character with zero (0) width. Spaces and punctuation are examples of neutral characters. Non-neutral characters are those with inherent directionality. For example, Roman letters (left-to-right) and Arabic letters (right-to-left) have strong directionality. In a "normal" Roman font where spaces and punctuation are present, the font direction hints should be set to two (2).

The unitsPerEm value determines the granularity of the font's coordinate grid at which coordinates can be specified for visual elements such as outline control points or mark-attachment anchor positions.

A glyph's bounding box is the smallest rectangle that contains all of the glyph's control points. (See 5.2.4 for more information.) The xMin, yMin, xMax and yMax values provide a bounding box that will contain all glyphs from the font. Glyphs without contours are ignored.

The macStyle bits shall agree with the fsSelection bits in the 'OS/2' table(5.1.8.20). The fsSelection bits are used over the macStyle bits in Windows. The PANOSE values and 'post' table values are ignored for determining bold or italic fonts.

For historical reasons, the fontRevision value contained in this table is not used by Windows to determine the version of a font. Instead, Windows evaluates the version string (ID 5) in the 'name' table.

The left sidebearing point for each glyph corresponds to the first "phantom" point ("pp1" below—see "Instructing TrueType Glyphs" [36]) and relates to the glyph left sidebearing (lsb) and xMin as follows:

$$pp1 = xMin - lsb$$

If bit 1 of the flags field is set, then $pp1 = 0$ for all glyphs and the xMin and left sidebearing for each glyph shall be equal.

In a variable font with TrueType outlines, the left side bearing for each glyph shall equal xMin, and bit 1 in the flags field shall be set. Also, bit 5 shall be cleared in all variable fonts. For general information on OFF Font Variations, see [subclause 7.1](#).

Also, in a variable font, the minimum or maximum x or y values of control points can vary, and a tight bounding rectangle containing the outline or all points for an instance of any given glyph can be smaller or larger than for the default instance of that glyph. The xMin, yMin, xMax and yMax values in this table might or might not encompass the derived glyph outlines for an instance. In addition, variation deltas are not provided for these values. If an application requires a bounding rectangle that encompasses the glyphs for a non-default instance of the font, the derived glyph outlines for the instance should be processed to determine a bounding rectangle.

5.1.4 hhea—Horizontal header

This table contains information for horizontal layout. The values in the minRightSidebearing, minLeftSidebearing and xMaxExtent should be computed using *only* glyphs that have contours. Glyphs with no contours should be ignored for the purposes of these calculations. All reserved areas shall be set to 0.

Type	Name	Description
uint16	majorVersion	Major version number of the horizontal header table — set to 1.
uint16	minorVersion	Minor version number of the horizontal header table — set to 0.
FWORD	ascender	Typographic ascent—see remarks below.
FWORD	descender	Typographic descent—see remarks below.
FWORD	lineGap	Typographic line gap. Negative lineGap values are treated as zero in some legacy platform implementations.
UWORD	advanceWidthMax	Maximum advance width value in 'hmtx' table.
FWORD	minLeftSidebearing	Minimum left sidebearing value in 'hmtx' table for glyphs with contours (empty glyphs should be ignored).
FWORD	minRightSidebearing	Minimum right sidebearing value; calculated as $\min(\text{aw} - (\text{lsb} + \text{xMax} - \text{xMin}))$ for glyphs with contours (empty glyphs should be ignored).
FWORD	xMaxExtent	$\text{Max}(\text{lsb} + (\text{xMax} - \text{xMin}))$.
int16	caretSlopeRise	Used to calculate the slope of the cursor (rise/run); 1 for vertical.
int16	caretSlopeRun	0 for vertical.
int16	caretOffset	The amount by which a slanted highlight on a glyph needs to be shifted to produce the best appearance. Set to 0 for non-slanted fonts

Type	Name	Description
int16	(reserved)	set to 0
int16	(reserved)	set to 0
int16	(reserved)	set to 0
int16	(reserved)	set to 0
int16	metricDataFormat	0 for current format.
uint16	numberOfHMetrics	Number of hMetric entries in 'hmtx' table

The ascender, descender and linegap values in this table are Apple specific; see the 'hhea' table specification of Apple's TrueType Reference Manual [7] for details regarding Apple platforms. The sTypoAscender, sTypoDescender and sTypoLineGap fields in the OS/2 table are used on the Windows platform, and are recommended for new text layout implementations. Font developers should evaluate behavior in target applications that can use fields in this table or in the OS/2 table to ensure consistent layout. See the descriptions of the OS/2 fields (5.1.8.23 – 5.1.8.25) for additional details.

'hhea' Table and OFF Font Variations

In a variable font, various font-metric values within the horizontal header table may need to be adjusted for different variation instances. Variation data for 'hhea' entries can be provided in the [metrics variations \('MVAR'\) table](#). Different 'hhea' entries are associated with particular variation data in the 'MVAR' table using value tags, as follows:

'hhea' entry	Tag
caretOffset	'hcof'
caretSlopeRise	'hcrs'
caretSlopeRun	'hcrn'

For general information on OFF Font Variations, see [subclause 7.1](#).

5.1.5 hmtx—Horizontal metrics

Glyph metrics used for horizontal text layout include glyph advance widths, side bearings and X-direction min and max values (xMin, xMax). These are derived using a combination of the glyph outline data ('glyf', 'CFF ' or 'CFF2') and the horizontal metrics table. The horizontal metrics ('hmtx') table provides glyph advance widths and left side bearings.

In a font with TrueType outline data, the '[glyf](#)' table provides xMin and xMax values, but not advance widths or side bearings. The advance width is always obtained from the 'hmtx' table. In some fonts, depending on the state of flags in the '[head](#)' table, the left side bearings may be the same as the xMin values in the 'glyf' table, though this is not true for all fonts. (See the description of bit 1 of the flags field in the 'head' table.) For this reason, left side bearings are provided in the 'hmtx' table. The right side bearing is always derived using advance width and left side bearing values from the 'hmtx' table, plus bounding-box information in the glyph description — see below for more details.