ISO/IEC JTC 1/SC 29/WG 03 N0492

**ISO/IEC JTC 1/SC 29/WG 03  
MPEG Systems   
Convenorship: KATS (Korea, Republic of)**

**Document type:** Output Document

**Title:** Technologies under Consideration for Open Font Format

**Status:** Approved

**Date of document:** 2022-01-27

**Source:** ISO/IEC JTC 1/SC 29/WG 03

**No. of pages:** 44 (with cover page)

**Email of Convenor:** young.L @ samsung . com

**Committee URL:** <https://isotc.iso.org/livelink/livelink/open/jtc1sc29wg3>

**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 03 MPEG SYSTEMS**

**ISO/IEC JTC 1/SC 29/WG 03 N0492**

**January 2022, Virtual**

|  |  |
| --- | --- |
| **Title** | **Technologies under Consideration for Open Font Format** |
| **Source** | **WG 03, MPEG Systems** |
| **Status** | **Approved** |
| **Serial Number** | **21213** |

This document summarizes proposes updates to OFF sub-clause 5.4.3 “CFF2 – Compact Font Format (version 2) table”.

The goal of these updates is to make CFF2-based fonts significantly easier to understand for developers of font tools, font rasterizers and any other code that deals with binary fonts. The updated proposal fixes errors, has more explanations, clearer tabulation, and has involved restructuring and rewriting of the entire CFF2 specification. It also removes dependencies on other documents, notably the Adobe Type 1 specification.

Notably, it *does not* change or add data structures, fields, operators, etc.

***Editor’s Note***: Highlighted text indicates subclause references that need to be verified when the content of this document is integrated with the rest of the OFF standard.

### **CFF2 – Compact Font Format (version 2) table**

#### 5.4.3.1 Overview

The CFF2 table is used for describing glyphs in an OFF font. It is an alternative to the ‘glyf’ table (subclause 5.3.4) using an efficient format to represent glyph outlines that has origins in the Adobe® Postscript® language.

In the CFF2 table, sequences of cubic (3rd-order) Bézier curves and straight lines are used to define glyph outlines. CFF2 data can also include “blend” operations, controlled by the font variations mechanism to change the shapes of glyphs. A rasterization fill rule is used to provide the opaque, monochrome shape of each glyph. CFF2 data can include “hint” operations that influence this rasterization. When combined with the COLR (subclause 5.7.11) and CPAL (subclause 5.7.12) tables, the CFF2 table can be used to represent multi-color glyphs.

CFF2 is the successor to the ‘CFF ’ table (subclause 5.4.2) and glyph format. Because of the origins of CFF in the Postscript language, the data in a complete ‘CFF ’ table can be used as a stand-alone font. When used within an OFF font, however, that aspect of CFF results in redundancy. CFF2 avoids redundancy by relying on data in other font tables.

A summary of significant differences between glyph representations in the ‘CFF2’, ‘CFF ’ and ‘glyf’ tables is given in subclause 5.4.3.14.

#### 5.4.3.2 Organization of the CFF2 table

The CFF2 table is comprised of various required and optional subtables. The following description summarizes the overall structure of the CFF2 table. The various subtables are described in subsequent parts of this document.

Certain subtables use a DICT structure—a binary dictionary format with one or more key-value pairs. The common DICT structure is described in subclause 5.4.3.7.

Certain other subtables use an INDEX structure, which contains one or more data objects of various types. The common INDEX structure in described in subclause 5.4.3.4.

The CFF2 table begins with a short header. The header is followed by the TopDICT subtable, which uses the DICT format and stores offsets to other subtables. The TopDICT subtable is followed by the GlobalSubrINDEX subtable, which uses the INDEX format and stores CharString data (see below) that can be re-used in multiple glyph descriptions.

The first three structures—Header, TopDICT and GlobalSubrINDEX—must occur in that order at the start of the CFF2 table. Other subtables may occur in any order at offsets indicated in the TopDICT subtable and elsewhere.

The CharStringINDEX subtable uses the INDEX format to store CharString data. A CharString is an encoded representation of a glyph, including the glyph outline data as well as hinting and variation data specific to the glyph. (See subclause 5.4.3.11 for details.) Within the CharStringINDEX, there is one CharString for each glyph in the font. CharStrings that have encoded data in common with other CharStrings may use Subrs (subroutines) to save space. Subroutines are stored in the GlobalSubrINDEX subtable or in PrivateSubrINDEX data blocks within a PrivateDICT subtable.

Each CharString is dependent on metadata related to hinting and subroutines that is stored in a FontDICT / PrivateDICT pair. (The metadata is stored in a PrivateDICT; a corresponding FontDICT provides the location of the PrivateDICT.) A CFF2 table only requires a single FontDICT / PrivateDICT pair. Multiple FontDICT / PrivateDICT pairs may be used, however. In particular, if some CharStrings require a different set of metadata from other CharStrings, then multiple FontDICT / PrivateDICT pairs may be defined. In such cases, a FontDICTSelect subtable is included to specify which FontDICT / PrivateDICT pair is used by each CharString.

Metadata relating to Font Variations is stored in a *VariationStore* subtable. If the font does not support Font Variations, then the VariationStore must be omitted.

*CF2F Data layout*

|  |  |  |
| --- | --- | --- |
| **Data Block** | **Required** | **Offset from start of CFF2 table** |
| Header | Yes | 0 |
| TopDICT | Yes | 5 |
| GlobalSubrINDEX | Yes | 5 + Header.TopDictLength |
| CharStringINDEX | Yes | TopDICT: CharStringINDEXOffset |
| FontDICTSelect | No | TopDICT: FontDICTSelectOffset |
| FontDICTINDEX | Yes | TopDICT: FontDICTINDEXOffset |
| FontDICT#0 | Yes | +FontDICTINDEX.offsets[0] |
| FontDICT#1 | - | +FontDICTINDEX.offsets[1] |
| … |  |  |
| FontDICT#n | - | +FontDICTINDEX.offsets[n] |
| PrivateDICT#0 | Yes | FontDICT#0: PrivateDICTOffset |
| PrivateDICT#1 | - | FontDICT#1: PrivateDICTOffset |
| … |  |  |
| PrivateDICT#n | - | FontDICT#n: PrivateDICTOffset |
| VariationStore | No | TopDICT: VariationStoreOffset |

#### 5.4.3.3 Numbers

**OFF Number formats**

The standard OFF number formats *uint8*, *int16*, *uint16*, *uint24*, *int32*, *uint32* and *Fixed* are used in certain CFF2 structures as specifed in subclause 4.3 “Data Types”.

**DICT and CharString number formats**

DICT and CharString data blocks (including subroutines) are intended to be decoded using a stack that uses the special number formats specified below. During decoding, bytes representing numbers and operators are encountered. Numbers have various encodings depending on their magnitude and on whether they are integers, and may in all cases be identified as numbers by their initial byte. Most number encodings may be used in both DICT and Charstring data, but note the exceptions listed.

The following table shows how numbers are encoded based on an initial byte **b0** interpreted as *uint8*, and possible subsequent bytes **b1**, **b2**, **b3**, etc. In the 2-byte formats, interpret b1 as *uint8*.

*Numerical Value Encoding*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initial byte b0** | **Value range** | **Value calculation** | **Size in bytes** | **Usage** |
| 32 to 246 | -107 to +107 | b0 - 139 | 1 | both |
| 247 to 250 | 108 to 1131 | (b0 - 247) \* 256 + b1 + 108 | 2 | both |
| 251 to 254 | -1131 to -108 | -(b0 - 251) \* 256 - b1 - 108 | 2 | both |
| 28 | -32768 to 32767 | interpret b1 and b2 as *int16* | 3 | both |
| 255 | -32768 to (32768 - 1/65536) | interpret b1 to b4 as *Fixed* | 5 | *CharString* only |
| 29 | -(2^31) to (2^31-1) | interpret b1 to b4 as *int32* | 5 | *DICT* only |
| 30 | any real number | following bytes are *Binary Coded Decimal* | unlimited | *DICT* only |

The *Usage* column denotes whether the format may be used in DICT data, in CharString data, or in both. Numbers encoded as int32 (initial byte 29) or Binary Coded Decimal (initial byte 30) are only permitted in DICT data. Numbers encoded as Fixed (initial byte 255) are only permitted in *CharString* data. It is not possible to represent numbers >= 32768 or < -32768 in *CharString* data.

If the initial byte b0 is not in this list (remembering that the valid list is different for *DICT* and *CharString* data), then it is either an operator or an error.

Once decoded, numbers are immediately pushed onto the stack for use as operands for forthcoming operators.

*Binary Coded Decimal*

Real numeric values of arbitrary precision can be represented in a binary coded decimal form. This representation uses a binary encoding of a decimal numeric expression, such as “123.456”. The numeric expression can optionally use decimal exponential notation, such as “1.23456 × 10^2”. Binary coded decimal numbers may only be used in DICT data.

The representation of binary coded decimal numbers begins with a prefix byte value of 30. This is followed by a byte sequence in which each 4-bit nibble represents an element. The two nibbles of each byte are interpreted in big-endian order: the first element is stored in the most significant 4 bits, and the second element is stored in the least significant 4 bits. The sequence is terminated with a nibble value of 0xf (hexadecimal).

After the prefix byte of 30 is recognized, the value of the binary coded decimal number is obtained by stepping through the nibbles, building up the decimal expression of the number, until the termination nibble is encountered. Each nibble value is interpreted according to the following table. Converted to ASCII, the string can then be converted to a high-precision floating point number using standard functions in most programming languages.

*Nibble Definitions*

|  |  |  |
| --- | --- | --- |
| **Nibble Value** | **Nibble Value (hex)** | **Represents in ASCII** |
| 0 to 9 | 0 to 9 | 0 to 9 |
| 10 | a | . (decimal point) |
| 11 | b | E |
| 12 | c | E– |
| 13 | d | <reserved> |
| 14 | e | – (minus) |
| 15 | f | end of number |

If the terminating 0xf nibble is the first nibble of a byte pair, then an additional 0xf is appended so that the encoded representation is always a whole number of bytes.

Negative exponents, as in the example 3×10^-5 (=0.00003), must be represented using the nibble value 0xc, not the nibble sequence 0xb followed by 0xe.

The maximum degree of precision obtainable from a Binary Coded Decimal is implementation dependent.

*Examples:*

* the value -2.25 is encoded by the byte 30 (decimal) followed by the byte sequence e2 a2 5f (hexadecimal)
* the value 0.140541E-3 is encoded by 30 (decimal) followed by 0a 14 05 41 c3 ff (hexadecimal).

The following regular expression validates a Binary Coded Decimal represented as ASCII:

-?([1-9][0-9]\*|0)?(\.[0-9]\*)?(E-?[1-9][0-9]\*)?

The following table shows some edge cases and their values:

|  |  |
| --- | --- |
| **Input** | **Value** |
| [empty] | 0 |
| . | 0 |
| .5 | 0.5 |
| 2. | 2 |
| e5 | *invalid* |
| 05 | *invalid* |
| e05 | *invalid* |

#### 5.4.3.4 INDEX data

*Used in: CharStringINDEX, GlobalSubrINDEX, LocalSubrINDEX, FontDICTINDEX*

An **INDEX** in CFF2 is a binary storage format for an array of arbitrary binary objects. Objects in the array are accessed by a 0-based index. Each object in the array may be of arbitrary size. Objects are stored contiguously and in order.

* INDEX data starts with *count*, the number of objects in the INDEX.
* The next field, *offSize*, specifies the number of bytes required to store each offset, that is, whether the offsets are *uint8* (1 byte), *uint16* (2 bytes), *uint24* (3 bytes) or *uint32* (4 bytes).
* Next comes the *offsets* array, which contains *count + 1* offset values, each of them in the number format specified by *offSize*. Offsets point to locations within the *data* block. Offset values are relative not to the start of the *data* block but to the byte preceding the *data* block (i.e., relative to the location of the last byte of the offsets array).
* Finally comes the *data* block itself, which stores all of the binary data objects themselves.

NOTE Because of the requirement that the *data* block stores binary objects are stored contiguously and in order, and the rule that the offset values are relative to the byte preceding the *data* block, the first element of the *offsets* array must be 1.

An object is retrieved by looking up an array index in the *offsets* array and fetching the binary data at the specified offset. The size of any object in bytes can be determined by subtracting its offset from the next offset in the *offsets* array. Since there are *count* + 1 offsets, this method works for all objects including the last object. Every object thus has a corresponding offset (which is always nonzero) and a size (which may be zero).

An empty INDEX is represented by a *count* field with a 0 value and no additional fields. Thus, the total size of an empty INDEX is 4 bytes.

The total size of a non-empty INDEX is:

4 + 1 + offSize \* (count + 1) + offsets[count] - 1

When creating INDEX data blocks, it is recommended to use the smallest possible representation for offsets. For example, if *offsets*[*count*] is greater than 255 and less than 65536, then all offsets can be represented as uint16, and so an *offSize* of 2 should be used.

The following table shows the layout of INDEX data.

*INDEX Data Format*

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| uint32 | count | Number of objects stored in INDEX |
| uint8 | offSize | Offset array element size (1, 2, 3 or 4) |
| uint8 *or* uint16 *or* uint24 *or* uint32 | offsets [count+1] | Array of offsets – offsets are from byte preceding object data. |
| uint8 | data [<unlimited>] | total length of *data* is the last offset - 1 |

#### 5.4.3.5 Decoding DICT and CharString data using a stack

Two of the fundamental data encoding schemes in CFF2 are the *DICT* and the *CharString*. Each is a binary data block that represents a sequence of encoded numbers and operators. In order to interpret those sequences, CFF2 uses the abstract data structure of a *stack*.

##### 5.4.3.5.1 Description of a stack

A stack is a kind of array conceived as a physical stack of items, which may be manipulated in only two ways:

1. place one new item on the top (“push an item to the stack”);
2. remove one item from the top (“pop an item from the stack”).

A CFF2 decoder is expected to implement a stack in a suitable manner. The CFF2 stack stores only numbers.

##### 5.4.3.5.2 The stack-based decoder in CFF2

Starting with an empty stack, a decoder of *DICT* and *CharString* data processes bytes sequentially from the start to the end, decoding *numbers* and *operators*.

***Numbers***

Numbers are represented using an encoding scheme of 1 or 2 bytes for integers >= -1131 and <= 1131, a range that is very common in fonts, and additional bytes for integers of larger magnitude and non-integers. For these number encodings, see subclause 5.4.3.3.

Decoded numbers are immediately *pushed* to the stack, ready for use as operands for forthcoming operators.

***Operators***

Operators are encoded with 1 or 2 bytes, specified so that they do not overlap with the encodings allocated for numbers. All two-byte operators start with the byte 0x0c. A few operators also encode data in succeeding bytes.

Decoded operators *pop* one or more numbers from the stack, then store or process them, and in some cases *push* new numbers back to the stack. The maximum number of operands on the CFF2 stack is 513.

The function of *DICT* operators is to create key-value pairs. During decoding, the *DICT* key is identified by the operator, and the *DICT* value is obtained by popping all the items from the stack (that is, one or more numbers), after which the stack is empty. The *DICT* operator *blend* is exceptional, neither creating a key-value pair nor leaving the stack empty. Instead, its function is to modify values before they are assigned to *DICT* keys. See subclause 5.4.3.13 for details on the *blend* operator.

The function of *CharString* operators is to decode data describing glyph outlines, hints and variations. During decoding, most *CharString* operators pop all the numbers on the stack and do not push any new numbers to the stack; thus *CharString* operators leave the stack empty. There are three exceptions:

* *blend*: pops operands from the stack (but not necessarily all the operands on the stack) and pushes processed numbers back to the stack;
* *callsubr*: pops a number from the stack which invokes a local subroutine (subroutines are not restricted regarding the state of the stack);
* *callgsubr*: invokes a global subroutine.

##### 5.4.3.5.3 Stack notation

In specifications for *DICT* and *CharString*, this document uses the following notation to describe the state of the stack when particular operators are encountered.

[ — represents the bottom of the stack

] — represents the top of the stack

x, dx, etc. — values on the stack referred to in the description

<number> — a number, either an integer or real number

<integer> — an integer

( ) — delimiters for a group of items that is repeated or optional

\* — an item or group occurs 0 or more times

+ — an item or group occurs 1 or more times

... — items at the bottom of the stack that are not involved in the current operation

##### 5.4.3.5.4 Examples

[ <number> ] — the stack contains one number

[ <integer> ] — the stack contains one integer

[ <number>+ ] — the stack contains 1 or more numbers

[ (<number> <number>)+ ] — the stack contains an even number of numbers (using the regular expression-style parentheses for grouping)

[... <number>\* <integer> ] — at the top of the stack are 0 or more numbers, followed by an integer ##

#### 5.4.3.6 Header

This data block contains 3 bytes for version specification and a uint16 representing the length of the TopDICT.

*Header Format*

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| uint8 | majorVersion | Format major version (set to 2). |
| uint8 | minorVersion | Format minor version (set to 0). |
| uint8 | headerSize | Header size (set to 5) |
| uint16 | topDictLength | Length of TopDICT structure. |

The TopDICT data starts immediately after the Header, thus at offset 5 from the start of the CFF2 table.

The HeaderSize value must be used when locating the start of the TopDICT data. It is provided so that future versions of the specification may increase the size of the Header without disabling older implementations.

The sum *HeaderSize + TopDICTLength* is the location of the required *GlobalSubrINDEX* subtable in the CFF2 table.

#### 5.4.3.7 DICT data

*Used in and with: TopDICT, FontDICT, PrivateDICT, FontDICTINDEX and FontDICTSelect*

A DICT subtable defines a “dictionary” data structure consisting of key-value pairs. DICTs are used in CFF2 to store offsets to other data blocks, and metadata used across multiple glyphs. In each key-value pair:

* The *key* is a string from a predefined vocabulary, and is encoded by an *operator* of one or two bytes. Each type of DICT subtable has a different set of keys which are valid. A specific key may not be defined more than once in any DICT.
* The *value* is a number or a sequence of numbers.

DICT data is decoded using a stack: see subclause 5.4.3.5. Starting from the beginning of the DICT data, numbers and operators are parsed from the data sequentially until the end of DICT data is reached. Decoded numbers are pushed onto the stack. DICT operators pop numbers from the stack, and assign those numbers to the key identified by the operator itself.

With the exception of the *blend* operator, after a DICT operator and its values have been decoded, the stack should be empty. If it is not empty, the subtable is invalid.

In general, the key-value pairs for a DICT may be specified in any order. However, some ordering rules are specified for *PrivateDICT*.

While decoding a DICT, if an operator is encountered that is not defined for the current DICT type, the behavior is unspecified.

It is permitted for multiple *DICT*s to share the same encoded data. In this way, one *DICT* could represent a subset of the data of another *DICT*.

##### 5.4.3.7.1 Offsets and lengths of DICT data blocks

The methods to determine the offset and the length of the three different types of DICT are given in the following table.

|  |  |  |
| --- | --- | --- |
| **DICT type** | **Offset** | **Length** |
| *TopDICT* | 5 | stored in *Header* as *TopDICTLength* |
| *FontDICT* | stored in *FontDICTINDEX* | stored in *FontDICTINDEX* |
| *PrivateDICT* | stored in the *PrivateDICTOffset* key of a *FontDICT* | stored in the *PrivateDICTOffset* key of a *FontDICT* |

#### 5.4.3.8 TopDICT

The *TopDICT* is a DICT that provides offsets to various parts of the CFF2 table, and also provides values relating to *unitsPerEm* in the *head* table (subclause 5.2.3). *TopDICT* data immediately follows *Header* data (see subclause 5.4.3.6) in a CFF2 table. Every CFF2 table requires one *TopDICT*. For a description of DICTs, see subclause 5.4.3.7.

##### 5.4.3.8.1 TopDICT operator summary

The following table lists the 5 operators allowed in a *TopDICT*, the bytes by which they are encoded in DICT data as hexadecimal and decimal, and whether or not they are required.

*Top DICT Operator Entries*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hex Value** | **Dec Value** | **Name** | **Required** | **Default** |
| 0x11 | 17 | *CharStringINDEXOffset* | yes | – |
| 0x18 | 24 | *VariationStoreOffset* | only for fonts with variations | – |
| 0x0c24 | 12,36 | *FontDICTINDEXOffset* | yes | – |
| 0x0c25 | 12,37 | *FontDICTSelectOffset* | no | – |
| 0x0c07 | 12,7 | FontMatrix | no | 0.001 0 0 0.001 0 0 |

##### 5.4.3.8.2 TopDICT operator specifications

The following specifies the 5 keys that are valid in a TopDICT.

*CharStringINDEXOffset: 0x11 (17)*

Stack: <integer>  
Required. This entry defines the offset to the *CharStringINDEX* data, from the start of the CFF2 table.

*VariationStoreOffset: 0x18 (24)*

Stack: <integer>  
Required if the font uses variations. Forbidden in non-variable fonts. This entry defines the offset to the *VariationStore* data, from the start of the CFF2 table.

*FontDICTINDEXOffset: 0x0c24 (12, 36)*

Stack: <integer>  
Required. This entry defines the offset to the *FontDICTINDEX* data, from the start of the CFF2 table.

*FontDICTSelectOffset: 0x0c25 (12, 37)*

Stack: <integer>  
Optional. This entry defines the offset to the *FontDICTSelect* data, from the start of the CFF2 table. If the CFF2 has only one FontDICT, and therefore no need for *FontDICTSelect* data, then *FontDICTSelectOffset* must not be defined.

*FontMatrix: 0x0c07 (12, 7)*

Stack: <number> 0 0 <number> 0 0  
Required if *unitsPerEm* is not equal to 1000. This entry defines the scale factor for glyph coordinates within the em square, similar to the *unitsPerEm* field in the *head* table (subclause 5.2.3). However, a reciprocal value is used (thus 1 / *unitsPerEm*) and this value occurs twice in the 6 operands. For the common case where *unitsPerEm* is 1000, a default key-value pair is created with the value “0.001 0 0 0.001 0 0” and the *FontMatrix* operator should be omitted.

The *Binary Coded Decimal* number format (see subclause 5.4.3.3) is used to represent 1 / *unitsPerEm*. For example, if *unitsPerEm* is 2000, then the value 0.0005 must be represented as Binary Coded Decimal.

NOTE The origin of the *TopDICT* *FontMatrix* key is a 2×3 transformation matrix. However, in CFF2 only matrices with uniform horizontal and vertical scaling without translation are permitted, thus the 1st and 4th entries must be identical and the remaining entries must be zero.

#### 5.4.3.9 FontDICT

A FontDICT is a DICT (see subclause 5.4.3.7) data block that provides offsets to a related PrivateDICT (subclause 5.4.3.10), which in turn provides metadata relating to hinting, variations and subroutines for a subset of CharStrings. Each CharString in a CFF2 table requires access to a PrivateDICT, and this is provided by assigning a FontDICT to each CharString.

A FontDICT exists only to point to a PrivateDICT. Hence there is only one operator allowed in a FontDICT, namely PrivateDICTOffset. FontDICTs are stored in FontDICTINDEX.

It is common for all CharStrings in a font to use the same PrivateDICT, and thus the same FontDICT. When there is only one FontDICT in a CFF2 table, all CharStrings are assigned to it. When there are multiple FontDICTs, the FontDICTSelect structure is required, in order to assign FontDICTs to CharStrings.

There is no specified limit to the number of FontDICTs.

*PrivateDICTOffset: 0x12 (18)*

Stack: <integer> <integer>  
Required. The two numbers are, respectively, the size and offset of the related *PrivateDICT* data block. The offset is from the start of the CFF2 table. This is the only operator allowed in a *FontDICT*. If the related *PrivateDICT* is empty, then the numbers 0 0 must be used.

##### 5.4.3.9.1 FontDICTINDEX

The FontDICTINDEX is an INDEX (subclause 5.4.3.4) data block that stores all the FontDICT data blocks in the CFF2 table. At least one FontDICT is required, therefore FontDICTINDEX requires at least one entry. The location of FontDICTINDEX in the CFF2 table is given by the FontDICTINDEXOffset key in TopDICT.

If the CFF2 table has more than one FontDICT, then a FontDICTSELECT structure is required, in order to assign glyphs to FontDICTs.

An upper limit to the number of FontDICT structures in a CFF2 table is not specified.

##### 5.4.3.9.2 FontDICTSelect

The FontDICTSelect data block assigns each CharString (i.e. each glyph) in the CFF2 table to a particular FontDICT in FontDICTINDEX. The location of FontDICTSelect in the CFF2 table is given by the FontDICTSelectOffset key in TopDICT.

In many fonts all CharStrings use the same FontDICT (and therefore the same PrivateDICT). Such fonts do not store FontDICTSelect data and their TopDICT omits the FontDICTSelectOffset key.

There are three formats defined for FontDICTSelect: format 0, format 3 and format 4. All formats provide numGlyphs mappings from a CharString identifier (gid) to a FontDICT identifier (fd), where numGlyphs is the number of CharStrings in CharStringINDEX. Formats 3 and 4 map ranges of gids onto a single fd, which often makes those formats a better choice for efficiency.

*Format 0*

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| uint8 | format | Set to 0 |
| uint8 | fds [numGlyphs] | *FontDICT* selector array |

Each element of the *fds* array represents the FD index of a Font DICT in the FDArray. This format should be used when the FD indices are in a fairly random order. The number of glyphs (numGlyphs) is the value of the count field in the CharStrings INDEX.

*Format 3*

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| uint8 | format | Set to 3 |
| uint16 | nRanges | Number of ranges |
| Range3 | range3[nRanges] | Array of Range3 records (see below) |
| uint16 | sentinel | Sentinel *gid* |

The format of a Range3 record is as follows:

*Range3 Record Format*

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| uint16 | first | First *gid* in range |
| uint8 | fd | *FontDICT* index for all glyphs in range |

Each *Range3* describes a group of sequential *gid*s that have the same *fd* index. Each range includes *gid*s from the first *gid* in the range record up to, but not including, the first *gid* of the next range record. Records in the *Range3* array must be in increasing order of first *gid*. The first range must have a first *gid* of 0. A sentinel *gid* follows the last range element and serves to delimit the last range in the array. The sentinel *gid* is set equal to *numGlyphs*, the number of glyphs in the font. That is, its value is 1 greater than the last *gid* in the font. The largest value for *fd* is 255.

NOTE Since the sentinel *gid* delimits the last range in the array, its value, encoded as a *uint16*, cannot exceed the value 65535. Therefore, the last *gid* encoded when using *FontDICTSelect* Format 3 cannot exceed 65534.

*Format 4*

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| uint8 | format | Set to 4 |
| uint32 | nRanges | Number of ranges |
| Range4 | range4[nRanges] | Array of Range4 records (see below) |
| uint32 | sentinel | Sentinel *gid* |

Format 4 is identical to Format 3, except that it accommodates more than 65534 glyphs by using a uint32 type for the nRanges and sentinel fields, and a Range4 record array, which accommodates FontDICTINDEXes of a length up to 65535.

The format of a Range4 record is as follows:

*Range4 Record Format*

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| uint32 | first | First *gid* in range |
| uint16 | fd | *FontDICT* index for all glyphs in range |

The Range4 format differs from the Range3 only in that it accommodates more than 65536 glyphs by using a uint32 for the first *gid* field, and accommodates more than 255 *FontDICTs* by using a *uint16* field for the *fd* index.

NOTE While *FontDICTSelect* format 4 allows for more than 65535 glyphs, other parts of the OFF format, such as the *numGlyphs* field of the ‘maxp’ table, are still constrained to 65535 glyphs.

#### 5.4.3.10 PrivateDICT

A PrivateDICT is a DICT (subclause 5.4.3.7) subtable providing metadata relating to hinting, subroutines and variations that are used by a subset of CharStrings. Many fonts use only one PrivateDICT, whose metadata therefore applies to all CharStrings. Access to a PrivateDICT, in terms of length and offset of the subtable, is provided by the PrivateDICTOffset key in a FontDICT.

When different subsets of CharStrings use different PrivateDICTs, multiple FontDICTs must be defined in FontDICTINDEX, each pointing to a PrivateDICT. In such cases a FontDICTSelect structure is also required in order to allocate FontDICTs to CharStrings. For more information on FontDICTs, see subclause 5.4.3.9.

A PrivateDICT may be empty, requiring no storage. In this case, the PrivateDICTOffset key of FontDICT must set size and offset to \*0\*, and default values will be used for vsindex, BlueScale, BlueShift, BlueFuzz, LanguageGroup and ExpansionFactor, as set out in the table below.

Some PrivateDICT values relating to hinting can undergo interpolation with font variations using the blend operator. These are identified as “blendable” in the table below.

NOTE Non-integer numbers in DICT data, such as the values for *BlueScale* and *ExpansionFactor*, are represented as *Binary Coded Decimal* bytes.

***When to use multiple PrivateDICTs***

A CFF2 table requires at least one pair of FontDICT and PrivateDICT (even if the PrivateDICT is empty), and many fonts only have one pair. However, it may be beneficial to use more than one FontDICT (and hence more than one PrivateDICT) when the glyphs in the font are of multiple distinct styles; for example, when the font contains a Latin set plus an Emoji set, or a Latin set plus a Korean Hangul set. In such cases, specifying hint metadata for each glyph style in separate PrivateDICT structures will likely allow more compact and efficient CharString encoding, with better use of hinting zones, smaller index values in callsubr subroutine calls, and fewer operands in blend operations — resulting in less data in total. For similar reasons, the facility for multiple FontDICT/PrivateDICT pairs allow easier merging of multiple fonts whose glyphs already have hints, variation data and subroutines.

##### 5.4.3.10.1 PrivateDICT operator summary

The following table lists all *PrivateDICT* operators alongside their encoding as hexadecimal and decimal bytes, their default value (if any), their general purpose, and whether they may be blended.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Hex Value** | **Dec Value** | **Name** | **Default** | **Purpose** | **Blendable** |
| 0x13 | 19 | LocalSubrINDEXOffset | – | subroutines | no |
| 0x16 | 22 | vsindex | 0 | variation | no |
| 0x17 | 23 | blend | – | variation | yes |
| 0x06 | 6 | BlueValues | – | hinting | yes |
| 0x07 | 7 | OtherBlues | – | hinting | yes |
| 0x08 | 8 | FamilyBlues | – | hinting | no |
| 0x09 | 9 | FamilyOtherBlues | – | hinting | no |
| 0x0c09 | 12,9 | BlueScale | 0.039625 | hinting | no |
| 0x0c0a | 12,10 | BlueShift | 7 | hinting | no |
| 0x0c0b | 12,11 | BlueFuzz | 1 | hinting | no |
| 0x0a | 10 | StdHW | – | hinting | yes |
| 0x0b | 11 | StdVW | – | hinting | yes |
| 0x0c0c | 12,12 | StemSnapH | – | hinting | yes |
| 0x0c0d | 12,13 | StemSnapV | – | hinting | yes |
| 0x0c11 | 12,17 | LanguageGroup | 0 | hinting | no |
| 0x0c12 | 12,18 | ExpansionFactor | 0.06 | hinting | no |

##### 5.4.3.10.2 Subroutine operator

***LocalSubrINDEXOffset****: 0x13 (19)*

*Stack*: [ <integer> ]  
*Defines a location for the local subroutines that CharStrings may access.*

This specifies the offset, from the start of the *PrivateDICT*, to the *LocalSubrINDEX*. The *LocalSubrINDEX* stores all the local subroutines that may be used by *CharString*s associated with this *PrivateDICT*, using the *callsubr* operator. If there are no local subroutines, then the *LocalSubrINDEXOffset* operator must be omitted. For more information on subroutines, see subclause 5.4.3.11.

##### 5.4.3.10.3 Variation operators

Two variation operators are defined that depend on various other parts of the CFF2 table and on other tables in the OFF font. For their full descriptions, see *Font Variations in CFF2* (subclause 5.4.3.13). A summary of their function is given below.

***vsindex****: 0x16 (22)*

*Stack*: [ <integer> ]  
*Activates a particular list of variation regions from VariationStore.*

When this operator is used in a *PrivateDICT*, then the active list of regions (also known as an *ItemVariationData*) is used not only within the *PrivateDICT*, but also for all *CharString*s that reference that *PrivateDICT*. However, a *vsindex* operator used inside a *CharString* always takes precedence.

NOTE The *vsindex* operator is encoded as 0x16 in *PrivateDICT* data, unlike in *CharString* data where it is encoded as 0x0f.

***blend****: 0x17 (23)*

*Stack*: [... <number>+ <integer> ]  
*Pops a certain number of operands from the stack, processes them according to the font variations interpolation algorithm, then pushes processed numbers back onto the stack.*

Blending hinting values is useful, for example, to interpolate *BlueZones* between those required for the default Regular instance and those required for the Bold instance of a font.

For the operation of *blend*, see CFF2 Variation Store operators (subclause 5.4.3.13.4).

NOTE The *blend* operator is encoded as 0x17 in *PrivateDICT* data, unlike in *CharString* data where it is encoded as 0x10.

##### 5.4.3.10.4 Hinting operators

***BlueValues****: 0x06 (6)*

*Stack*: [ <deltaArray> ]  
*Font-wide vertical alignment zones.*  
*Blendable: yes*

The value represented by *BlueValues* is an array containing an even number of integers taken in pairs, and which follow a small number of rules:

* The first integer in each pair is less than or equal to the second integer in that pair.
* The first pair is the baseline overshoot position and the baseline. This is a bottom-zone.
* All subsequent pairs describe top-zones, that is, alignment zones for the tops of character features, for example, x-height and x-height overshoot position, ascender-height and ascender-height overshoot position, cap-height and cap-height overshoot position, figure-height and figure-height overshoot position.
* Up to seven pairs may be given in the *BlueValues* array; the first pair must be the baseline pair.
* Different pairs must be at least 3 units apart from each other and from pairs in *OtherBlues*, as described in the following section. (This minimum distance can be modified by the optional *BlueFuzz* entry in the *PrivateDICT*; see the definition of *BlueFuzz*, that follows.)
* The maximum difference between values in one pair is constrained as described under the description of *BlueScale*, that follows.

Despite the names often given to the various alignment zones described by the *BlueValues*, renderers have no built-in notions of which parameters apply to which glyphs. Each zone helps to control the alignment of any and all characters with character level hints that fall within the zone.

***Example***

Consider the following array that represents three alignment zones in a typeface, being baseline (at 0 with negative overshoot at -15), cap-height (at 700 with overshoot at 715) and x-height (at 547 with overshoot at 559), thus:

[-15 0 700 715 547 559]

However, because this must be stored as a *deltaArray*, the encoded numbers in the *PrivateDICT* would be, in fact:

[-15 15 700 15 -168 12]

Notice that this saves 2 bytes compared with storing the absolute numbers, because smaller numbers are encoded with fewer bytes in DICT data.

***OtherBlues****: 0x07 (7)*

*Stack*: [ <deltaArray> ]  
*Additional bottom alignment zones.*  
*Blendable: yes*

The optional *OtherBlues* entry in the *PrivateDICT* is associated with an array of pairs of integers similar to the *BlueValues* array. However, the *OtherBlues* array describes bottom-zones only. For example, these may include: descender-depth overshoot position and descender-depth, superior baseline overshoot position and superior baseline, and ordinal baseline overshoot position and ordinal baseline. Up to five pairs (10 integers) may be specified in the *OtherBlues* array. Numbers in a pair must be in ascending order, with the same restriction on the maximum difference in a pair. Pairs must be at least 3 units apart from all other pairs, including those in the *BlueValues* array. (This minimum distance can be modified by the optional *BlueFuzz* entry in the *PrivateDICT*.)

***FamilyBlues****: 0x08 (8)*

*Stack*: [ <deltaArray> ]  
*Family-wide vertical alignment zones.*  
*Blendable: no*

When different styles of the same font family are mixed in text, it is often desirable to coordinate their x-heights, cap-heights, and other alignments so that they will be the same at small sizes. For example, at 72 pixels per inch, the x-height of a 10-point roman face might be 5.4 pixels while the boldface x-height might be 5.6 pixels. If the roman face is the regular for the family, the renderer can render both faces with an x-height of 5 pixels instead of letting the boldface jump to 6 while the roman is still at 5. However, at 100 points, the roman x-height will be 54 pixels and the bold x-height will be 56.

You can include information about the dominant alignment zones in a font family so that this consistency can be enforced. When enabled, if the difference between a font’s alignment and its family’s standard alignment is less than 1 pixel, then the renderer will use the standard alignment instead of the normal alignment for that font. Thus at 10 points in the previous example, the difference is 5.6 − 5.4 = 0.2 pixels so the standard is used. At 100 points, the difference is 56 − 54 = 2, so the specific x-height for the font is used. Family alignment values are identical to individual font alignment values; i.e., they are things like x-height, x-height overshoot, etc. The *PrivateDICT* entries are as follows.

* The value associated with *FamilyBlues* is an array containing an even number of integers taken in pairs. The rules governing the contents of this array are analogous to those of the *BlueValues* array.
* The value associated with *FamilyOtherBlues* is an array containing an even number of integers taken in pairs. The rules governing the contents of this array are analogous to those of the *OtherBlues* array.

Typically, the *FamilyBlues* and *FamilyOtherBlues* entries will simply be copied from the *BlueValues* and *OtherBlues* of the regular face in the family. Each font in a family (except the regular) must have these entries if it is to have family alignment properties. Of course, if these entries are not present, then only a font’s own alignment hints will be considered.

***FamilyOtherBlues****: 0x09 (9)*

*Stack*: [ <deltaArray> ]  
*Family-wide bottom alignment zones.*  
*Blendable: no*

See *FamilyBlues* for description.

***BlueScale****: 0x0c09 (12,9)*

*Stack*: [ <number> ]  
*Related to point size at which to deactivate overshoot suppression.* *Blendable: no*

The optional *BlueScale* entry in the *PrivateDICT* controls the point size at which overshoot suppression ceases. This point size varies with the number of device pixels per inch available on the device where the font is being rendered.

* For point sizes that occupy fewer device pixels than the *BlueScale* value results in for a given device, overshoot suppression is performed. All features falling in an alignment zone are rendered at the same pixel height.
* For point sizes that occupy the same number or a greater number of device pixels than the *BlueScale* value results in, overshoot suppression is turned off, thus allowing overshoots to occur. (This behavior may be modified by the optional *BlueShift* setting; see the definition of *BlueShift*, that follows.)

The *BlueScale* value is a number directly related to the number of pixels tall that one character space unit will be before overshoot suppression is turned off. The default value of *BlueScale* is 0.039625, which corresponds to 10 points at 300 dpi. (Thus if that value is acceptable, a *PrivateDICT* does not need to define *BlueScale*.) A simple formula that relates point size as rendered on a 300-dpi device to the *BlueScale* value is:

*BlueScale* = (*pointsize* − 0.49) / 240

The formula provides a convenient number that font producers can use to determine at what integer point size overshoot suppression should be off. However, the exact point size at which overshoot suppression ceases is actually 0.49 points less (at 9.51 points using the default value of BlueScale) than the value of *pointsize* used in the formula. Adobe recommends using the adjustment shown in the formula so that the change in overshoot suppression behavior occurs at an exact point size unlikely to be used in practice.

For example, if you wish overshoot suppression to turn off at 11 points on a 300-dpi device, you should set *BlueScale* to (11 − 0.49) / 240 or 0.04379. With this one setting of *BlueScale*, overshoot suppression will turn off at proportionately smaller point sizes on higher resolution output devices or larger point sizes on lower-resolution devices such as displays.

***BlueShift****: 0x0c0a (12,10)*

*Stack*: [ <integer> ]  
*Overshoot enforcement. Optional, but relevant even if Flex is not used.*  
*Blendable: no*

The optional *BlueShift* entry in the *PrivateDICT* adds another capability to the treatment of overshoot behavior. The value of *BlueShift* is an integer that indicates a character space distance beyond the flat position of alignment zones at which overshoot enforcement for character features occurs. The default value of *BlueShift* is 7. The single setting of *BlueShift* applies to all alignment zones, regardless of where their over- shoot positions lie.

When a character’s size is less than that expressed by *BlueScale*, character features that fall within alignment zones have their overshoots suppressed. For characters larger than the *BlueScale* size, character features that fall beyond the flat position of an alignment zone (above for top-zones, below for bottom-zones) by a character space distance equal to or greater than the value of *BlueShift* will overshoot, while character features closer to the flat position than the *BlueShift* value will overshoot only if their device space distance is at least one-half pixel.

If one or more *flex* operators occurs in any *CharStrings* using the current *PrivateDICT* (that is, any of *flex*, *hflex*, *hflex1*, *flex1*), then the *BlueShift* value must be greater than *flex depth*. Since the default value of *BlueShift* is 7, this entry must be set explicitly if *flex depth* is greater than 6. For example, if *flex depth* is 8, then set *BlueShift* to 9. If *flex depth* is 6 or less, then *BlueShift* may be omitted.

***BlueFuzz****: 0x0c0b (12,11)*

*Stack*: [ <integer> ]  
*Extends the range of alignment zones. Optional.*  
*Blendable: no*

The optional *BlueFuzz* entry in the *PrivateDICT* is an integer value that specifies the number of character space units to extend (in both directions) the effect of an alignment zone on a horizontal stem. If the top of a horizontal stem is within *BlueFuzz* units (in character space) outside of a top-zone, the interpreter will act as if the stem top were actually within the zone; the same holds for the bottoms of horizontal stems in bottom-zones. The default value of *BlueFuzz* is 1. *BlueFuzz* has been a convenient means for compensating for slightly inaccurate coordinate data. The effect of a non-zero value for *BlueFuzz* can usually be better achieved by adjusting the sizes of the alignment zones. Adobe suggests that new fonts not rely on it and disable the feature by explicitly setting *BlueFuzz* to 0 in the *PrivateDICT*.

NOTE Because a non-zero value for *BlueFuzz* extends the range of alignment zones, alignment zones must be declared at least (2 × *BlueFuzz* + 1) units apart from each other. Therefore, a default *BlueFuzz* value of 1 implies that alignment zones should be at least 3 units apart from each other.

***StdHW****: 0x0a (10)*

*Stack*: [ <number> ]  
*Dominant horizontal stem width.*  
*Blendable: yes*

***StdVW****: 0x0b (11)*

*Stack*: [ <number> ]  
*Dominant vertical stem width.*  
*Blendable: yes*

***StemSnapH****: 0x0c0c (12,12)*

*Stack*: [ <deltaArray> ]  
*Array of common horizontal stem widths.*  
*Blendable: yes*

***StemSnapV****: 0x0c0d (12,13)*

*Stack*: [ <deltaArray> ]  
*Array of common vertical stem widths.*  
*Blendable: yes*

Stem Width Information, controlled by these 4 operators, is a mechanism to tell the renderer about standard stem widths in a font so that the renderer can ensure consistency at small sizes. If a particular stem is slightly wider or narrower than standard, either by design or as a result of a small error in creating the font, then at small sizes where a single pixel difference would be very noticeable, the renderer can render the stem as though it had the standard width. However, at large sizes where a single pixel difference will produce only a subtle visual effect, the stem will be allowed to deviate from the standard.

When the difference between a standard stem width and a particular stem width is small, the standard width is used. For example, if at 10 points a standard stem width corresponds to 1.4 pixels wide and a particular stem is 1.6 pixels wide, both can be rendered as a 1-pixel wide stem. However, at 100 points the standard stem would be rendered as 14 pixels wide and the particular stem would be rendered as 16 pixels wide. The information that the renderer needs appears in the following *PrivateDICT* entries.

* *StdHW* takes a number expressing the dominant width of horizontal stems (measured vertically in FUnits).
* *StdVW* takes a number expressing the dominant width of vertical stems (measured horizontally in FUnits). Typically, this will be the width of straight stems in lower case letters. (For an italic font, give the width of the vertical stem measured at an angle perpendicular to the stem direction.)
* *StemSnapH* is an array of up to 12 numbers of the most common widths (including the dominant width given in *StdHW*) for horizontal stems (measured vertically). These widths must be sorted in increasing order.
* *StemSnapV* is an array of up to 12 numbers of the most common widths (including the dominant width given in the *StdVW*) for vertical stems (measured horizontally). These widths must be sorted in increasing order. For example, you might include widths for straight and curved stems in upper and lower case letters. For an italic font, this array should be empty.

If these Stem Width Information hints are not present in the *PrivateDICT*, then each stem is rendered according to its own definition in the *CharString* (as modified by any other *CharString* hints).

***LanguageGroup****: 0x0c11 (12,17)*

*Stack*: [ <integer> ]  
*Identifies language group of font.*  
*Blendable: no*

Certain groups of written languages share broad aesthetic characteristics. Identification of such language groups can prove useful for accurate character rendering.

The value of the entry *LanguageGroup* is an integer that indicates the language group of the *CharString*s (glyphs) using this *PrivateDICT*. If the *PrivateDICT* does not contain this entry, or if the given value is not recognized, then the value of *LanguageGroup* defaults to zero.

Two language groups are defined:

* group 0 consists of languages that use Latin, Greek, Cyrillic, and similar alphabets. Since the value of the *LanguageGroup* entry defaults to 0, a *PrivateDICT* containing glyphs corresponding to one of these languages does not need to contain this entry.
* group 1 consists of Chinese ideographs and similar character sets, including Japanese Kanji and Korean Hangul. A *PrivateDICT* that contains glyphs corresponding to one of these languages should define *LanguageGroup*.

***ExpansionFactor****: 0x0c12 (12,18)*

*Stack*: [ <number> ]  
*Provides control over rendering of counters.*  
*Blendable: no*

The optional *ExpansionFactor* entry is a real number that gives a limit for changing the size of a character bounding box during the processing that adjusts the sizes of counters in fonts of *LanguageGroup* 1. The default value of *ExpansionFactor* is 0.06. At small point sizes or low resolutions, the system may have to accept irregular counters rather than violate this limit. Bar code fonts or logos that need counter control may benefit by setting *LanguageGroup* to 1 and increasing the *ExpansionFactor* limit to a larger amount such as 0.5 or more.

##### 5.4.3.10.5 Operator ordering restrictions

The *OtherBlues* operator, if used, must occur after the *BlueValues* operator.

The *FamilyOtherBlues* operator, if used, must occur after the *FamilyBlues* operator.

#### 5.4.3.11 CharString

The *CharString* format provides a method for compact encoding of glyphs, namely *path data*, *hint data* and *variation data*. The CFF2 *CharString* specification updates previous definitions of *CharString* and is intended for use only with a CFF2 font table in an OFF font file.

Like DICT data (subclause 5.4.3.7), the *CharString* is a binary data block that consists of a sequence of encoded *numbers* and *operators*, and is intended to be decoded using a stack. See subclause 5.4.3.5 for details.

##### 5.4.3.11.1 CharStringINDEX

All *CharString*s in a CFF2 table are stored in the *CharStringINDEX* data block. This is located at the offset specified by the *CharStringINDEXOffset* key in *TopDICT*. Indices in *CharStringINDEX* are used universally as glyph identifiers elsewhere in the OFF specification.

##### 5.4.3.11.2 CharString operator summary

With three exceptions, every operator processes all operands from the stack and leaves the stack empty. Thus, arguments are generally supplied only for the next operator. The exceptions are the variation operator *blend* and the subroutine operators *callsubr* and *callgsubr*, which all process only a single item or a limited number of items at the top of the stack. If an unrecognized operator is encountered, the behavior is unspecified.

The following table lists all *CharString* operators alongside their encoding as hexadecimal and decimal bytes, the type of operator, and whether or not it clears the stack. All 2-byte operators start with the byte 0x0c (12). For convenience the bytes introducing numbers are also shown; for the full specification of numbers in *CharString*s, see subclause 5.4.3.3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hex Value** | **Dec Value** | **Operator** | **Type** | **Clears stack** |
| 0x01 | 1 | hstem | hinting | yes |
| 0x03 | 3 | vstem | hinting | yes |
| 0x04 | 4 | vmoveto | path | yes |
| 0x05 | 5 | rlineto | path | yes |
| 0x06 | 6 | hlineto | path | yes |
| 0x07 | 7 | vlineto | path | yes |
| 0x08 | 8 | rrcurveto | path | yes |
| 0x0a | 10 | callsubr | subroutine | **no** |
| 0x0f | 15 | vsindex | variation | yes |
| 0x10 | 16 | blend | variation | **no** |
| 0x12 | 18 | hstemhm | hinting | yes |
| 0x13 | 19 | hintmask | hinting | yes |
| 0x14 | 20 | cntrmask | hinting | yes |
| 0x15 | 21 | rmoveto | path | yes |
| 0x16 | 22 | hmoveto | path | yes |
| 0x17 | 23 | vstemhm | hinting | yes |
| 0x18 | 24 | rcurveline | path | yes |
| 0x19 | 25 | rlinecurve | path | yes |
| 0x1a | 26 | vvcurveto | path | yes |
| 0x1b | 27 | hhcurveto | path | yes |
| 0x1d | 29 | callgsubr | subroutine | **no** |
| 0x1e | 30 | vhcurveto | path | yes |
| 0x1f | 31 | hvcurveto | path | yes |
| 0x0c22 | 12,34 | hflex | path | yes |
| 0x0c23 | 12,35 | flex | path | yes |
| 0x0c24 | 12,36 | hflex1 | path | yes |
| 0x0c25 | 12,37 | flex1 | path | yes |
| 0x1c | 28 | int16 value | number | — |
| 0x20..0xf6 | -107..107 | 1-byte number | number | — |
| 0xf7..0xfa | 108..1131 | 2-byte number | number | — |
| 0xfb..0xfe | -1131..-108 | 2-byte number | number | — |
| 0xff | 255 | Fixed format value | number | — |

##### 5.4.3.11.3 CharString organization

A complete *CharString*, including its subroutines, has the following structure:

* (*hstem* or *hstemhm*)\*  
  Any number of horizontal stem hints, followed by
* (*vstem* or *vstemhm*)\*  
  Any number of vertical stem hints, followed by
* (*cntrmask*)\*  
  Any number of counter mask hints, followed by
* *path data*  
  A path consisting of any number of contours. Each contour consists of one of the *moveto* operators followed by any number of line and curve operators. Within the path data there can be any number of *hintmask* operators.

The *blend* operator may be used in all of the path operators and in all of the hinting operators except *hintmask* and *cntrmask*. If the *vsindex* operator is used, it must be used before the first *blend* operator. Only one set of *VariationRegion*s may be used within a particular *CharString*.

Any part of *CharString* data may be packaged as a *subroutine*.

##### 5.4.3.11.4 Subroutines

A *subroutine* is a portion of *CharString* data that is invoked (or “called”) from within other *CharString* data. Multiple *CharString*s may efficiently call the same *subroutine*, thereby saving significant amounts of space.

An example of a *subroutine* could be data that describes a diacritic or serif shape that is identical in multiple glyphs. In practice, the identification of portions of *CharString* data suitable for becoming *subroutine*s is usually automated. There are no restrictions on how *CharString* data may be split into *subroutine*s; *subroutine*s do not need to represent complete contours, or even a complete set of operands with their operator.

There are two types of *subroutine*: \* global *subroutine*s are stored in *GlobalSubrINDEX* and may be called by any *CharString*; \* local *subroutine*s are stored in a *LocalSubrINDEX* and may be called only by *CharString*s that have access to that *LocalSubrINDEX* via their *PrivateDICT*.

A single *CharString* may use both types of *subroutine*.

Every CFF2 table includes a *GlobalSubrINDEX*. If there are no global *subroutine*s to store, then an empty *GlobalSubrINDEX* is required. See subclause 5.4.3.4 for decoding and storing empty INDEX structures.

CFF2 compilers use subroutines in order to minimize file size. However, using subroutines is not mandatory. Avoiding subroutines when creating CFF2 tables may result in faster encoding and potentially a smaller file size in subsequent file compression steps.

**Nesting subroutines**

Nesting of subroutines (one subroutine calling another subroutine) is allowed up to a maximum of 10 levels. Recursion (one *subroutine* calling itself, or any other *subroutine* in its nested *subroutine*s) is not allowed.

**Subroutine bias**

The operand supplied to a *callsubr* or *callgsubr* operator must be added to the *subroutine* *bias* number before being used as an index into *LocalSubrINDEX* or *GlobalSubrINDEX*. The *bias* is calculated from *nSubrs*, the number of *subroutine*s in the given INDEX:

* If *nSubrs* is in the range [0,1239] then *bias* = 107.
* If *nSubrs* is in the range [1240,33899] then *bias* = 1131.
* If *nSubrs* is >= 33900 then *bias* = 32768.

This technique allows *subroutine* identifiers to be specified using negative as well as positive numbers, efficiently utilizing the available number ranges.

**Subroutines and the stack**

The *subroutine* operators *callsubr* and *callgsubr*, unlike most other operators, do not clear the stack. For example, several numbers can be pushed onto the stack, then a *subroutine* call occurs, and the *subroutine* pops the operands for a path drawing operation.

**Implementing subroutines with a “return stack”**

Implementing subroutines in a CFF2 decoder requires keeping track of the location to which the *subroutine* should return after it has completed. The return location is in fact the byte immediately after that of the operator (*callsubr* or *callgsubr*) that called the *subroutine*. Because of potential nesting, it is recommended to implement a “return stack” so that multiple return locations can be stored. Care must be taken to check if the return location is beyond the end of the *CharString* or *subroutine* data from which the *subroutine* was called, in which case the decoder must perform one or more extra “return” steps (if it was called from another *subroutine*) or terminate decoding (if it was called from a *CharString*).

##### 5.4.3.11.5 Paths

**Summary**

A path decoded from a *CharString* consists of one or more contours, where each contour consists of a start point followed by one or more line segments and cubic Bézier curves. Line segments are defined by an implicit start point and provided *x* and *y* coordinates. Curves are defined by an implicit start point and 3 provided sets of *x* and *y* coordinates.

**Decoding paths**

Decoding *CharString* data into a path involves keeping track of the “current point”, initially at *(0, 0)*, and processing three basic types of operator:

* *moveto* operators start a new contour at a provided point;
* *lineto* operators define a line segment from the current point to a provided point;
* *curveto* operators define a cubic Bézier curve which starts at the current point, has two provided off-curve control points, and a provided end-point.

All *moveto*, *lineto* and *curveto* operators use relative coordinates: each point is defined as a relative coordinate offset from the point before it. This rule also applies to the three points of a curve operation, where the first provided point is relative to the current point, the second is relative to the first, and the third is relative to the second. After any path construction operation, the current point is then set to the last of the given points.

Operators for move, line and curve exist in multiple forms to avoid explicit representation of the number 0, which is a common relative measurement in paths. Also, the number of items on the stack can trigger multiple sequential path operations of the same type. Some operators combine curve and line specifications in a single operation.

**Path direction**

A contour that is to be filled must be defined in a counterclockwise orientation. A contour that is to be left unfilled must be defined in a clockwise orientation. If you imagine walking along a contour in the direction it is defined, then a filled area should be on your left.

NOTE These orientations are opposite to those specified in the *‘glyf’* table (subclause 5.3.4).

**Overlapping contours**

Overlapping contours are supported in CFF2 fonts. CFF2 renderers must use the non-zero winding number rule, rather than the even-odd rule. Thus one counter-clockwise contour completely inside another counter-clockwise contour does *not* introduce any transparency; both counters are rendered opaque. This behavior is necessary in order to support variable font data, where overlapping contours are common. Care must be taken therefore when converting CFF2 *CharString* data into CFF *CharString* data (or any other outline format that uses the even-odd rendering rule), for example by using an overlap removal operation.

**Non-printing glyphs**

Non-printing glyphs, such as the space character, require no path, hints or variation data. Therefore, no *CharString* data is stored. In the *CharStringINDEX*, the offset for such an empty *CharString* is identical to the offset for the subsequent *CharString*.

NOTE Horizontal and vertical metrics for non-printing glyphs are defined and varied using the usual methods.

**Contours that end with a curve**

Unlike contours in the *‘glyf’* table (subclause 5.3.4), CFF2 *CharString* contours are *always* self-closed with a straight line segment, even if it is of zero length. Thus, if the last explicit operand in a contour is a curve, a line segment of zero length is automatically inserted between it and the contour’s start-point. In static fonts this implicit line segment is invisible and causes no difficulty. However, in variable fonts, care must be taken when the start-point of a contour is blended, since the contour’s final explicit point does not necessarily move in the same way.

For example, consider a glyph contour representing an approximation of a circle. This can be constructed from 4 cubic Bézier curves, thus 12 control points. In CFF2, we may represent this contour as one *rmoveto* operation (with 2 operands) followed by four *rrcurveto* operations (each with 6 operands). Notice that 26 operands are used in total rather than 24, the last two denoting the same *x* and *y* coordinates as the start-point defined by *rmoveto*. When variations for this contour are defined using the *blend* operator, particular care must be taken to match the movement between the start-point and the last point of the last *rrcurveto*; otherwise, a line segment will appear to be inserted into the contour, linking the last point to the start-point.

**Implementation limits**

The total number of contours in a glyph is unlimited. The number of line and curve operations in a contour is also unlimited.

**Flex**

The four *flex* operators define Bézier curves that could be represented with two *rrcurveto* operators, except that they also define interaction with the rasterizer such that the curve should — when it is very small in device coordinates — be treated as a straight line.

**Resolution**

Coordinates for path operations are typically integers. Fixed 16.16 format coordinates are permitted, but require significantly more *CharString* data.

##### 5.4.3.11.6 Hints

In a *CharString*, hints specify horizontal and vertical regions that should be treated with special techniques by the rasterizer, resulting in a more regular appearance. Hints must be specified in the following order:

* horizontal stems and edges using the *hstem* or *hstemhm* operators,
* vertical stems and edges using the *vstem* or *vstemhm* operators,
* counters (gaps between stems), specified in terms of stems with the *cntrmask* operator.

Each of these types of hint is optional. However, if a counter hint is provided, it requires two stems in its orientation to be defined.

In some glyphs two or more stems overlap, potentially causing problems for a rasterizer deciding how to control outline points on the pixel grid. In such cases, the *hstemhm* and *vstemhm* stem definition operators must be used instead of *hstem* and *vstem*. Then the *hintmask* operator is used at various points within path construction data to control which hints are active for subsequent path construction operators.

##### 5.4.3.11.7 Subroutine operators

***callsubr****: 0x0a (10)*

*Stack*: [... <integer>]  
Call the subroutine in *LocalSubrINDEX* with the index determined by the operand. Note that the operand must be added to the subroutine bias number before being used as the index. See *Subroutine bias* for calculation of this number.

The *LocalSubrINDEX* key in the *PrivateDICT* of this *CharString* provides the location of the *LocalSubrINDEX*.

If *callsubr* is used within a global subroutine, care must be taken to invoke the local subroutine in the *LocalSubrINDEX* of the *CharString* being processed.

***callgsubr****: 0x1d (29)*

*Stack*: [... <integer>]  
Call the subroutine in *GlobalSubrINDEX* with the index determined by the operand. Note that the operand must be added to the subroutine bias number before being used as the index. See *Subroutine bias* for calculation of this number.

NOTE The *callsubr* and *callgsubr* operators pop from the stack the operand needed to determine the *subroutine* index, and leave the stack otherwise unchanged. The *subroutine* itself may pop and push operands freely.

##### 5.4.3.11.8 Path operators that move the current point and start a new contour

***rmoveto****: 0x15 (21)*

Stack: [ dx dy ]  
Moves the current point to a position at the relative coordinates *(dx, dy)* and starts a new contour.

***hmoveto****: 0x16 (22)*

Stack: [ dx ]  
Moves the current point *dx* units in the horizontal direction and starts a new contour.

***vmoveto****: 0x04 (4)*

Stack: [ dy ]  
Moves the current point *dy* units in the vertical direction and starts a new contour.

##### 5.4.3.11.9 Path operators that draw line segments

***rlineto****: 0x05 (5)*

Stack: [ (dx dy)+ ]  
Appends a line segment from the current point to a position at the relative coordinates *(dx, dy)*. Additional *rlineto* operations are performed for all subsequent argument pairs. The number of lines is determined from the number of arguments on the stack.

***hlineto****: 0x06 (6)*

Stack: [ (d)+ ]  
Appends a horizontal line segment from the current point to a position at the relative coordinates *(d, 0)*. When multiple arguments are used, the orientation alternates for each argument, thus the second argument is a vertical line to the relative coordinates *(0, d)*, the third argument is a horizontal line to the relative coordinates *(d, 0)*, and so on. The number of line segments is determined from the number of arguments on the stack. A contour that consists of only horizontal and vertical lines can be constructed using one of the *moveto* operators and a single *hlineto* operator, with the number of line segments limited only by the size of the number stack.

***vlineto****: 0x07 (7)*

Stack: [ (d)+ ]  
Appends a vertical line segment from the current point to a position at the relative coordinates *(0, d)*. As with *hlineto*, when multiple arguments are used, the orientation alternates for each argument.

##### 5.4.3.11.10 Path operators that draw curves

***rrcurveto****: 0x08 (8)*

Stack: [ (dxa dya dxb dyb dxc dyc)+ ]  
Appends a cubic Bézier curve, defined by the points *p0*, *p1*, *p2*, *p3* where *p0* is located at the current point, *p1* is given by the relative coordinates *(dxa, dya)*, *p2* is given by the relative coordinates *(dxb, dyb)* and *p3* is given by the relative coordinates *(dxc, dyc)*. The location of each point is defined relative to the preceding one. For each subsequent set of six arguments, an additional curve is appended to the current point. The number of curve segments is determined from the number of arguments on the number stack and is limited only by the size of the number stack.

***hhcurveto****: 0x1b (27)*

Stack: [ dy1? (dxa dxb dyb dxc)+ ]  
Appends one or more Bézier curves, as described by the *dxa…dxc* set(s) of arguments, to the current point. If the argument count is a multiple of four, the curve starts and ends horizontal. If the argument count is one more than a multiple of four, the first curve does not begin with a horizontal tangent and its *dy1* value is given by the first argument. This operator is the horizontal counterpart to *vvcurveto*.

***vvcurveto****: 0x1a (26)*

Stack: [ dx1? (dya dxb dyb dyc)+ ]  
Appends one or more Bézier curves, as described by the *dya…dyc* set(s) of arguments, to the current point. If the argument count is a multiple of four, the curve starts and ends vertical. If the argument count is one more than a multiple of four, the first curve does not begin with a vertical tangent and its *dx1* value is given by the first argument. This operator is the vertical counterpart to *hhcurveto*.

***hvcurveto****: 0x1f (31)*

Stack: [ dx1 dx2 dy2 dy3 (dya dxb dyb dxc dxd dxe dye dyf)\* dxf? ]  
Stack: [ (dxa dxb dyb dyc dyd dxe dye dxf)+ dyf? ]  
Appends one or more Bézier curves to the current point with tangents that alternate between horizontal and vertical. The first curve’s first tangent is horizontal and its second tangent is vertical. If a second curve is defined, its first tangent is vertical and its second tangent is horizontal. A third curve, if defined, is oriented like the first. The tangent of the final curve can be overridden freely by using the optional extra argument to define a non-zero coordinate. The equivalent operator that starts with a vertical tangent is *vhcurveto*.

***vhcurveto****: 0x1e (30)*

Stack: [ dy1 dx2 dy2 dx3 (dxa dxb dyb dyc dyd dxe dye dxf)\* dyf? ]  
Stack: [ (dya dxb dyb dxc dxd dxe dye dyf)+ dxf? ]  
Appends one or more Bézier curves to the current point with tangents that alternate between vertical and horizontal. The first curve’s first tangent is vertical and its second tangent is horizontal. If a second curve is defined, its first tangent is horizontal and its second tangent is vertical. A third curve, if defined, is oriented like the first. The tangent of the final curve can be overridden freely by using the optional extra argument to define a non-zero coordinate. The equivalent operator that starts with a horizontal tangent is *hvcurveto*.

***rcurveline****: 0x18 (24)*

Stack: [ (dxa dya dxb dyb dxc dyc)+ dxd dyd ]  
Appends a sequence of Bézier curves followed by a line. This is equivalent to one *rrcurveto* for each set of six arguments *dxa…dyc*, followed by exactly one *rlineto* using the *dxd, dyd* arguments. The number of curves is determined from the count on the argument stack.

***rlinecurve****: 0x19 (25)*

Stack: [ (dxa dya)+ dxb dyb dxc dyc dxd dyd ]  
Appends a sequence of lines followed by a Bézier curve. This is equivalent to one *rlineto* for each pair of arguments preceding the six arguments *dxb…dyd* needed for the one *rrcurveto* operation. The number of lines is determined from the count of items on the argument stack.

***flex****: 0x0c23 (12,35)*

Stack: [ dx1 dy1 dx2 dy2 dx3 dy3 dx4 dy4 dx5 dy5 dx6 dy6 fd ]  
Causes two Bézier curves, as described by the arguments (as shown in Figure 1 below), to be rendered as a straight line when the *flex depth* is less than *fd/100* device pixels, and as curved lines when the flex depth is greater than or equal to *fd/100* device pixels. The *flex depth* for a horizontal curve, as shown in Figure 2, is the distance from the join point to the line connecting the start and end points on the curve. If the curve is not exactly horizontal or vertical, determination of *flex depth* uses a method that depends on whether the curve is more horizontal or vertical by the method described in the *flex1* description, below, and as illustrated in Figure 2.

|  |
| --- |
|  |
| **Figure 1 – Flex hint example** |
|  |
| **Figure 2 – Flex depth calculations** |

NOTE In cases where some of the points have the same *x* or *y* coordinate as other points in the curves, arguments may be omitted by using one of the following forms of the *flex* operator: *hflex*, *hflex1* or *flex1*.

***hflex****: 0x0c22 (12,34)*

Stack: [ dx1 dx2 dy2 dx3 dx4 dx5 dx6 ]  
Causes the two curves described by the arguments *dx1…dx6* to be rendered as a straight line when the *flex depth* is less than 0.5 device pixels (that is, *fd* < 50), and as curved lines when the *flex depth* is greater than or equal to 0.5 device pixels.

***hflex*** is used when the following are all true:

* The starting point, the ending point, the first control point and the last control point all have the same *y* value (that is, *y* = *dy1* = *dy5* = *dy6*).
* The joining point and the neighboring control points have the same *y* value (that is, *dy2* = *dy3* = *dy4*).
* The flex depth is 50.

***hflex1****: 0x0c24 (12,36)*

Stack: [ dx1 dy1 dx2 dy2 dx3 dx4 dx5 dy5 dx6 ]  
Causes the two curves described by the arguments to be rendered as a straight line when the *flex depth* is less than 0.5 device pixels, and as curved lines when the flex depth is greater than or equal to 0.5 device pixels.

***hflex1*** is used if the conditions for *hflex* are not met but all of the following are true:

* The starting and ending points have the same *y* value.
* The joining point and the neighboring control points have the same *y* value.
* The flex depth is 50.

***flex1****: 0x0c25 (12,37)*

Stack: [ dx1 dy1 dx2 dy2 dx3 dy3 dx4 dy4 dx5 dy5 d6 ]  
Causes the two curves described by the arguments to be rendered as a straight line when the flex depth is less than 0.5 device pixels, and as curved lines when the flex depth is greater than or equal to 0.5 device pixels.

The *d6* argument will be either a *dx* or *dy* value, depending on the curve (see Figure 3). To determine the correct value, compute the distance from the starting point *(x, y)*, the first point of the first curve, to the last flex control point *(dx5, dy5)* by summing all the arguments except *d6*; call this *(dx, dy)*. If *abs(dx)* > *abs(dy)*, then the last point’s x-value is given by *d6*, and its y-value is equal to *y*. Otherwise, the last point’s x-value is equal to *x* and its y-value is given by *d6*.

*flex1* is used if the conditions for *hflex* and *hflex1* are not met but all of the following are true:

* The starting and ending points have the same *x* or *y* value.
* The flex depth is 50.

##### 5.4.3.11.11 Variation operators

***vsindex****: 0x0f (15)*

Stack: [ <integer> ]  
*Activates a particular list of variation regions from VariationStore.*

When used in a *CharString*, *vsindex* sets the active list of variation regions (also known as an *ItemVariationData*), overriding both the default list (index 0) and any list of regions that may be defined (using the *PrivateDICT* version of the *vsindex* operator) in the *PrivateDICT* of this this *CharString*.

NOTE The *vsindex* operator is encoded as 0x0f in *CharString* data, unlike in *PrivateDICT* data where it is encoded as 0x16.

***blend****: 0x10 (16)*

Stack: [... <n default numbers> <n \* k delta numbers> <n> ]  
*Pops n + (n\*k) + 1 operands from the stack, processes them according to the font variations interpolation algorithm, then pushes n processed numbers back onto the stack.*

For the operation of *blend*, see CFF2 Variation Store operators (subclause 5.4.3.13.4).

NOTE The *blend* operator is encoded as 0x10 in *CharString* data, unlike in *PrivateDICT* data where it is encoded as 0x17.

##### 5.4.3.11.12 Hint operators

The operators *hstem* and *vstem* respectively define horizontal and vertical hints for stems and edges, using numbers provided in pairs.

Each horizontal stem is defined by its bottom and top *y* coordinates. Each vertical stem is defined by its left and right *x* coordinates. All numbers are relative to the previous number, and the first number is relative to 0. Stems are defined in increasing order of bottom coordinate (for horizontal stems) or left coordinate (for vertical stems).

Edge hints are defined as if they were stem hints with negative thickness:

* a thickness of -21 defines a left edge or bottom edge;
* a thickness of -20 defines a top edge or right edge.

For the purposes of sorting, the first coordinate implied by an edge hint is considered its bottom or left coordinate.

If any of the hint definitions overlap in either the horizontal or vertical direction, then the operators *hstemhm* and *vstemhm* must be used for all hints in the *CharString* instead of *hstem* and *vstem*. The *hintmask* operator is used to select which hints are active at various points in the *CharString* data.

The *vstem* and *vstemhm* operators have the unusual property that in special cases they may be omitted from the *CharString* to save data. The special cases require both horizontal and vertical stems to be defined, and for those definitions to be immediately followed by a *hintmask* or *cntrmask* operator.

*Example*

Consider a sans-serif glyph E with the cap-height of 700 and with three horizontal stems whose *y* coordinates are: 0 and 80, 310 and 390, 620 and 700. These stems would be defined as:

0 80 230 80 230 80 hstem

In the same font, we also want to control the parts of other glyphs that align vertically with the stems of the E, but do not have any horizontal stems. The glyph I is often such a case, and benefits from two edge hints in the form of stems whose *y* coordinates are: 21 and 0, 700 and 680. Its bottom and top edges can be specified thus:

21 -21 700 -20 hstem

***hstem****: 0x01 (1)*

Stack: [ (<number> <number>)+ ]

Defines a sequence of non-overlapping horizontal stems for the *CharString* using pairs of numbers.

***vstem****: 0x03 (3)*

Stack: [ (<number> <number>)+ ]

Defines a sequence of non-overlapping vertical stems for the *CharString* using pairs of numbers.

***hstemhm****: 0x12 (18)*

Stack: [ (<number> <number>)+ ]

Defines a sequence of potentially overlapping horizontal stems for the *CharString* using pairs of numbers.

***vstemhm****: 0x17 (23)*

Stack: [ (<number> <number>)+ ]

Defines a sequence of potentially overlapping vertical stems for the *CharString* using pairs of numbers.

***hintmask****: 0x13 (19)*

Stack: [ ]

Activates and deactivates stem hints within this *CharString*, in order that overlapping hints do not cause ambiguities for the rasterizer.

If any hints overlap, *hintmask* must be used to establish a non-overlapping subset of hints for a portion of *CharString* data. Path operators occurring after a *hintmask* are influenced by the new set of active hints. The *hintmask* operator can be used any number times within one *CharString*.

*hintmask* does not pop any operands from the stack. Instead it consumes bytes that come after it in the *CharString* data, interpreting them as a bit field that flags each stem as active or inactive. The position of a bit in the bit field determines the index of the stem it controls, treating all horizontal and vertical stem as part of a single array of stems. The number of bytes used depends on the total number of horizontal and vertical stem, *numStems*, and is given by the formula: 1 + *floor* ( ( *numStems* - 1) /8 ). All unused bits must be 0.

*hintmask* may not be used unless the CharString also defines stems using *hstemhm* and/or *vstemhm*.

*Example*

A *hintmask* activating hints 1, 3 and 9 in a *CharString* that has 17 hints in total requires the bit field: 01010000 01000000 00000000. Thus the three bytes 0x504000 immediately follow the *hintmask* operator.

***cntrmask****: 0x14 (20)*

Stack: [ ]

Specifies the counter spaces to be controlled, and their relative priority.

Counter definitions depend on stem definitions (already defined using *hstem*, *vstem*, *hstemhm* and *vstemhm*). Counters are defined by specifying groups of stems that delimit them. More than two stems may be specified in order to delimit multiple counters.

Multiple *cntrmask* operators may be used where sets of counters are to be treated with different priority. Counters controlled by the first *cntrmask* have top priority; subsequent *cntrmask* commands specify lower priority counters.

*cntrmask* does not pop any operands from the stack. Instead it consumes bytes that come after it that flag stems as active or inactive. See *hintmask* for the formatting of these bytes.

Any *cntrmask* operators in a *CharString* must be placed immediately after the stem hint definitions. A *CharString* without defined stems may not use *cntrmask*.

##### 5.4.3.11.13 Metrics

*CharString*s do not contain any representation of horizontal or vertical glyph metrics. Metrics are stored in the *hmtx* (subclause 5.2.5) and *vmtx* (subclause 5.7.10) tables and undergo font variations using the *HVAR* (subclause 7.3.5) and *VVAR* (subclause 7.3.8) tables.

#### 5.4.3.12 Hinting in CFF2 fonts

This information is split between hinting in *PrivateDICT*s and hinting in *CharString*s

Please refer to the description of PrivateDICT (subclause 5.4.3.10) and CharString (subclause 5.4.3.11) for details.

#### 5.4.3.13 Font Variations in CFF2

##### 5.4.3.13.1 Introduction

The CFF2 table supports font variations as described in the chapter *Font Variations Overview* (subclause 7.1), and this section assumes familiarity with that chapter. In CFF2 tables, variation axis settings may adjust:

* positions of outline points defined in *CharString*s,
* hinting values defined in *CharString*s,
* hinting values defined in *PrivateDICT*s.

In general, to support variation of glyphs or other font data, font variations require a set of default values for the particular data item, a set of delta adjustment values used to modify the default values, and a set of regions within the font’s variation space over which the different delta values apply. In CFF2, the calculation of new values for particular instances — based on default values, delta values, regions and current axis settings — is called *blending*.

Blending is defined only for *CharString* and *PrivateDICT* data. In each case, two operators control variation:

* *vsindex* selects the active list of variation regions,
* *blend* updates numbers on the stack using delta values also provided on the stack.

The function of these operators is explained below.

A CFF2 table supporting font variations requires a *VariationStore* structure. This defines a global list of all the variation regions used in the font, and also multiple subsets of that list.

##### 5.4.3.13.2 VariationStore

The *VariationStore* starts with a *uint16* field that specifies a length, followed by an *Item Variation Store* structure of the specified length.

|  |  |  |
| --- | --- | --- |
| **Offset** | **Type** | **Field** |
| 0 | uint16 | *length* (of *Item Variation Store*) |
| 2 | <*length*> bytes | *Item Variation Store* |

For the specification of *Item Variation Store*, see *Font Variations Common Table Formats* (subclause 7.2). A brief summary is provided below.

**Summary of Item Variation Store**

An *Item Variation Store* consists of two arrays: an array of *VariationRegion*s and an array of *ItemVariationData* structures.

* A *VariationRegion* is defined in terms of three locations on each of the variation axes in the font. The three locations are the start, the peak and the end of the active region in each axis, all recorded as normalized axis values (i.e. between -1 and 1).
* The *VariationRegionList* is the complete list of *VariationRegion*s used in the CFF2 table. All the *VariationRegion*s used in the CFF2 font must be defined, even if they are used only in a single *CharString* or a single *PrivateDICT*. A CFF2 table that supports variations has exactly one *VariationRegionList*, which contains at least one *VariationRegion*.
* An *ItemVariationData* structure defines a *regionIndexes* array. Each item in *regionIndexes* identifies a *VariationRegion* via its 0-based index in *VariationRegionList*. (In the discussion of the *blend* operator, the length of the active *regionIndexes* array is referred to as *k*.)
* The *Item Variation Store* contains an array of *ItemVariationData* structures. Often the *Item Variation Store* contains only one *ItemVariationData* structure, hence only one set of regions is used by all glyphs.

**Other considerations for the CFF2 Variation Store**

Although the *ItemVariationData* structure in general allows the specification of delta values within itself, CFF2 does not use this facility. Instead, in CFF2 delta values are stored as operands to the *blend* operator. Therefore, the *itemCount* and *shortDeltaCount* fields in each *ItemVariationData* must be set to 0. Since *itemCount* is 0, there are no *deltaSets* to store, so the *deltaSets* field is not stored at all.

Like all *Item Variation Store*s, that used in CFF2 *VariationStore* is dependent on the *fvar* (subclause 7.3.3) and *avar* (subclause 7.3.1) tables that define the font’s variation space.

The location of the *VariationStore* in the CFF2 table is given by the *VariationStoreOffset* key of *TopDICT*. If there is no such key in a CFF2 table, the variations are not defined, the *VariationStore* data block must be omitted, and the *blend* operator must not be used.

NOTE The CFF2 *VariationStore* structure does not share data with *Item Variation Store* structures in other parts of the font. For example, if the *VariationRegionList* used in the CFF2 table is identical with that in the *MVAR* table, it must be defined explicitly in both places.

##### 5.4.3.13.3 CFF2 Variation Store operators

**The *vsindex* operator**

***vsindex****: 0x16 (22) in PrivateDICT data, 0x0f (15) in CharString data*

Stack: [ <integer> ]  
*Activates a particular list of VariationRegions from VariationStore.*

If multiple *ItemVariationData* structures are defined, then *PrivateDICT*s and *CharString*s need to be able to select an active list of *VariationRegion*s. The operator *vsindex* is provided to select the active *ItemVariationData* via its index in *Item Variation Store*, and hence sets the active list of regions. The default *vsindex* is 0, thus fonts with only a single list of regions have no need for the *vsindex* operator.

When a *PrivateDICT* specifies an active list of *VariationRegion*s using the *vsindex* operator, it applies not only to any blending that occurs within the *PrivateDICT* but also to blending in all of the *CharString*s that use that *PrivateDICT*. However, using the *vsindex* operator in a *CharString* overrides the set of regions set using *PrivateDICT*. Within a given *PrivateDICT* or *CharString* only one list of regions is active throughout, thus multiple occurrences of *vsindex* in a *PrivateDICT* or *CharString* are forbidden. If used in a *CharString*, then *vsindex* must occur before any use of *blend*.

**The *blend* operator**

***blend****: 0x17 (23) in PrivateDICT data, 0x10 (16) in CharString data*

Stack: [... <n default numbers> <n \* k delta values> <n> ]  
*Pops n + (n\*k) + 1 operands from the stack, processes them according to the font variations interpolation algorithm, then pushes n “blended” numbers back onto the stack (where n is the integer at the top of the stack, and k is the number of VariationRegions in the active ItemVariationData).*

The *blend* operator is CFF2’s mechanism for interpolating default values with delta values. Interpolation happens in the contexts of:

* an active *ItemVariationData* structure that provides a list of *k VariationRegions* (the active *ItemVariationData* structure is controlled using the *vsindex* operator);
* a set of normalized axis settings based on axis settings selected by a user or system.

Based on the axis settings and the configuration of the *VariationRegions*, the Font Variations system supplies a scalar (a real number in the range [0,1]) for each of the *k* *VariationRegions*, yielding an array of *k* scalars.

The *n+(n\*k)+1* arguments to the *blend* operator are organized in three groups:

* *n* default values;
* *n* \* *k* delta values, organized into *n* groups each of *k* delta values, one group for each of the *n* default values;
* *n* itself.

To calculate the blended values, the *n* default values are treated in turn. The *k* delta values for a particular default value are respectively multiplied by the *k* scalars. These *k* products are then all added to the default value, resulting in its final blended value. Once all *n* default values have been blended, they are pushed onto the stack for use by other operators.

*Examples*

Consider a value of 120 used in a *hlineto* operator that has a delta of 52 FUnits in the context of an *ItemVariationData* that has just one variation region active (*k* = 1). After the single default value (120) we also require *n* \* *k* delta values, and finally *n* itself. Since *n* =1 and *k* =1, the number of delta values is 1 \* 1 = 1. So the complete sequence of numbers and operators is as follows:

120 52 1 blend hlineto

For an instance where axis settings mean that the scalar for the one active region is 0.75, we can calculate the blended value by multiplying the delta value by that scalar, and then adding it to the default value, thus in this case 120 + 52 \* 0.75 = 159. So in this instance, we effectively have a *hlineto* line segment as if a static font had the following sequence of numbers and operators:

159 hlineto

**Advanced uses of *blend***

Although the *blend* operator always works in conjunction with a subsequent operator expecting precalculated values (such as *rlineto* or *BlueValues*), and is ultimately required to leave operands on the stack ready for that static operator, it is not required to process all of those operands in a single *blend* operation. Thus the number *n* in a *blend* need not equal the number of operands expected by the subsequent static operator, *nStatic*.

When *n* < *nStatic*, only a subset of the operands for the static operator are blended. For example, the two operands to *rlineto* would normally be blended in a single *blend* with *n*=2, but using two separate *blend* operations, each with *n*=1 and each placed immediately after the relevant operand, is also valid. Also, if only one of the operands has variation deltas, then a *blend* with *n*=1 immediately after that operand can save some data.

Furthermore, by means of multiple *blend* operations before a static operator and also by pushing more values onto the stack than might be expected, values already blended once can be blended multiple times. Only in the very last *blend* operation directly before the static operator are the blended offsets added to the default operands. This technique can be used to provide higher order interpolation using a single variation axis, for example to control the variation of a point along a quadratic, cubic or quartic curve rather than along a straight line.

**Why permit multiple lists of variation regions?**

It may reasonably be asked why the *Item Variation Store* structure provides access to the global list of variation regions only via multiple *ItemVariationData* structures that each contains a list of indices into the global list. Why not keep all variation regions permanently active?

Indeed, fonts that have consistent variation behavior in all glyphs should use a single *ItemVariationData* structure, whose *regionIndexes* array simply relists all the *VariationRegion* items in the *VariationRegionList*.

However, fonts that have heterogeneous variation behavior across their glyphs benefit from multiple *ItemVariationData* structures. Consider a 1-axis font where a special glyph uses 20 intermediate regions, while 100 other glyphs use only 1 region based on default and maximum axis settings. Thus 21 *VariationRegion*s must be specified in *VariationRegionList*. Yet if all 21 regions exist in a single *ItemVariationData* structure, then the *blend* operations in the 100 glyphs, in addition to the delta that controls their intended variation, must also specify 20 deltas with value 0 for all the intermediate regions to ensure they are inactive. With two *ItemVariationData* structures (the first containing 1 region, the second containing 20 regions) the special glyph can use the *vsindex* operator to select the second list of regions, and all *blend* operators can avoid using deltas of 0. In a large font, the data saved by this technique can be considerable.

##### 5.4.3.13.4 Comparison of CFF2 variations with ‘gvar’ variations

**Metrics**

When processing fonts that use a *gvar* table for font variations, the rasterizer appends four “phantom points” to the array of glyph points, based on metrics data from the *hmtx* (subclause 5.2.6) and *vmtx* (subclause 5.7.10) tables. These phantom points represent glyph metrics, and they can then be varied using *gvar* deltas as if they were points on the glyph outline. The CFF2 rasterizer also uses the *hmtx* and *vmtx* tables for metrics, but it does not create phantom points for manipulation in *CharString*s. Instead CFF2 depends on the *HVAR* (subclause 7.3.5) and *VVAR* (subclause 7.3.8) tables for variation of glyph metrics.

**Variation regions**

In *gvar*, a global array of variation regions is defined in the *sharedTuples* array. A record for each glyph defines a custom array of regions, where each region is either selected from the global array or defined only for that glyph. CFF2, by requiring all regions to be defined globally, and also by predefining the order of regions when referenced in a *blend* operation, requires less data overall for specifying regions.

**Inferred deltas**

The *gvar* variations mechanism includes functionality for “inferred deltas”. By moving only certain points on a glyph outline, deltas can be inferred for other points, saving data overall. See Inferred deltas for un-referenced point numbers (subclause 7.3.4.4).   
*Inferred deltas are not supported in CFF2 fonts.*

**Moving contours**

Although CFF2 lacks functionality for inferred deltas, an entire contour may still be efficiently translated by variable *x* and *y* offsets by applying variation deltas to a single point on the contour to be moved. In *gvar*, any point on the contour may be chosen, the other points moving identically via inferred deltas. In CFF2, since all points of a contour are defined relative to the contour’s initial *rmoveto* operation, it is only the operands of that *rmoveto* that should be *blended*. The locations of subsequent points on the contour, having deltas of zero relative to the initial point, can avoid *blend* operations entirely. Care must be taken to disable unintended variation on subsequent contours in the *CharString*, by means of opposing deltas on the operands of the subsequent *rmoveto*.

#### 5.4.3.14 Comparing CFF2 with CFF and glyf

The three formats intended for storing monochrome glyph outlines in OFF fonts are the ‘glyf’ table (subclause 5.3.4), the ‘CFF’ table (subclause 5.4.2), and the CFF2 table.

CFF2 and CFF use cubic (3rd order) Bézier curves to represent glyph outlines, whereas the ‘glyf’ table uses quadratic (2nd order) Bézier curves. CFF2 and CFF also use a different conceptual model for “hints” than the ‘glyf’ table. The three tables also differ in relation to support of variations and in how variation data is stored.

The following table provides a summary comparison of the CFF2, ‘CFF’ and ‘glyf’ tables. Note that some of these differences might not be exposed in high-level font editing software or in runtime programming interfaces.

|  |  |  |  |
| --- | --- | --- | --- |
| **Consideration** | **glyf** | **CFF** | **CFF2** |
| *curves* | quadratic (2nd order) | cubic (3rd order) | cubic (3rd order) |
| *coordinate precision* | 1 FUnit | 1/65536 FUnit | 1/65536 FUnit |
| *hinting* | TrueType instructions move outline points by controlled amounts | alignment zones apply to all glyphs, stem locations are declared in each glyph | alignment zones apply to all glyphs, stem locations are declared in each glyph |
| *decoding* | not stack-based (except TrueType instructions) | mostly stack-based | mostly stack-based |
| *Font variations* | yes: outline variation data is stored in ‘gvar’ (subclause 7.3.4); hint variation data is stored in ‘cvar’ (subclause 7.3.2) | no | yes: variation data for outlines and hints is stored within the CFF2 table |
| *data redundancy* | low | moderate | low |
| *overlapping contours* | yes | no | yes |

#### 5.4.3.15 CFF2 dependencies on other OFF data

An important factor in the design of the CFF2 specification was the avoidance of recording data already specified elsewhere in an OFF font file. Consequently, CFF2 is highly dependent on data in other OFF tables. OFF tables that do not refer to specific glyph formats are intended to function as specified, including ‘cmap‘ (subclause 5.2.2), ‘hhea‘ (subclause 5.2.4), ‘name‘ (subclause 5.2.7), ‘post‘ (subclause 5.2.10), ‘OS/2‘ (subclause 5.2.8), ‘GPOS‘ (subclause 6.3.3), ‘GSUB‘ (subclause 6.3.4), the color tables ‘COLR‘ (subclause 5.7.11) and ‘CPAL‘ (subclause 5.7.12), and the variation tables ‘fvar‘ (subclause 7.3.3), ‘avar‘ (subclause 7.3.1), ‘STAT‘ (subclause 7.3.7), ‘MVAR‘ (subclause 7.3.6), ‘HVAR‘ (subclause 7.3.5) and ‘VVAR‘ (subclause 7.3.8).

Here follow some requirements and considerations for specific tables.

##### 5.4.3.15.1 SFNT Header

**sfntVersion**

The *sfntVersion* field, the first four bytes of an OFF font file, must be 0x4F54544F (“OTTO”) for OFF fonts with glyphs defined in a CFF2 table.

##### 5.4.3.15.2 ‘maxp’ table

**version**

CFF2 fonts must use *version* 0.5 (encoded as 0x00005000)

**numGlyphs**

This must be identical to the number of *CharString*s in the CFF2 table (the number of items in *CharStringsINDEX*).

This version of the *maxp* table is only 6 bytes in total and avoids specifying values not used by CFF2.

##### 5.4.3.15.3 ‘head’ table

The specification should be followed for all fields, with the following notes.

**unitsPerEm**

This is respected, and must correspond with the value of the *TopDICT* FontMatrix key in the CFF2 table. FontMatrix uses the reciprocal of *unitsPerEm* in a specific sequence of operands. For example, if *unitsPerEm* is 2000, having reciprocal 1/2000 = 0.0005, then FontMatrix must be set to 0.0005 0 0 0.0005 0 0. If *unitsPerEm* is equal to 1000, then the FontMatrix key may be omitted from the *TopDICT*.

**indexToLocFormat**

Set to 0.

**glyphDataFormat**

Set to 0.

##### 5.4.3.15.4 ‘hmtx’ and ‘HVAR’ tables

Vertical metrics for each glyph are stored in the ‘hmtx’ table (subclause 5.2.5). Variation of horizontal metrics is controlled by the ‘HVAR’ table (subclause 7.3.5).

##### 5.4.3.15.5 ‘vmtx’ and ‘VVAR’ tables

Vertical metrics for each glyph are stored in the ‘vmtx’ table (subclause 5.7.10). Variation of vertical metrics is controlled by the ‘VVAR’ table (subclause 7.3.8).

##### 5.4.3.15.6 ‘fvar’ table

An ‘fvar’ (subclause 7.3.3) table is required if a CFF2 table uses font variations. In other words, if a *VariationStore* structure (subclause 7.2.3) is defined within the CFF2 table, an ‘fvar’ table must also be defined. This table specifies the number of axes, *axisCount*, as well as their order. For each axis it specifies the fields *axisTag*, *nameID*, *defaultValue*, *minValue*, and *maxValue*, which are all necessary for any implementation of font variations.

The CFF2 VariationStore data structure (see subclause 5.4.3.13.2) and the *blend* operators refer to variation axes according to their ordering in the ‘fvar’ table.

##### 5.4.3.15.7 ‘gvar’ table

The ‘gvar’ table (subclause 7.3.4) is not used in OFF fonts with CFF2-encoded glyphs. Variation in glyph outlines and glyph hinting are controlled directly in CharString data. Variation of glyph metrics is controlled in the ‘HVAR’ (subclause 7.3.5) and ‘VVAR’ tables (subclause 7.3.8).

# Annex: Example CFF2 Font

This annex illustrates the CFF2 format with an example CFF2 table.

Binary dump (226 bytes):

0000: 02 00 05 00 07 CF 0C 24 C3 11 9B 18 00 00 00 00  
0010: 00 26 00 01 00 00 00 0C 00 01 00 00 00 1C 00 01  
0020: 00 02 C0 00 E0 00 00 00 C0 00 C0 00 E0 00 00 00  
0030: 00 00 00 02 00 00 00 01 00 00 00 02 01 01 03 05  
0040: 20 0A 20 0A 00 00 00 01 01 01 05 F7 06 DA 12 77  
0050: 9F F8 6C 9D AE 9A F4 9A 95 9F B3 9F 8B 8B 8B 8B  
0060: 85 9A 8B 8B 97 73 8B 8B 8C 80 8B 8B 8B 8D 8B 8B  
0070: 8C 8A 8B 8B 97 17 06 FB 8E 95 86 9D 8B 8B 8D 17  
0080: 07 77 9F F8 6D 9D AD 9A F3 9A 95 9F B3 9F 08 FB  
0090: 8D 95 09 1E A0 37 5F 0C 09 8B 0C 0B C2 6E 9E 8C  
00A0: 17 0A DB 57 F7 02 8C 17 0B B3 9A 77 9F 82 8A 8D  
00B0: 17 0C 0C DB 95 57 F7 02 85 8B 8D 17 0C 0D F7 06  
00C0: 13 00 00 00 01 01 01 1B BD BD EF 8C 10 8B 15 F8  
00D0: 88 27 FB 5C 8C 10 06 F8 88 07 FC 88 EF F7 5C 8C  
00E0: 10 06

Analysis:

|  |  |  |
| --- | --- | --- |
| **Hex data** | **Source** | **Comments** |
| 02 | majorVersion | = 2 |
| 00 | minorVersion | = 0 |
| 05 | headerSize | = 5 |
| 00 07 | topDictSize | = 7 |
| CF 0C 24 | [ offset ] **FontDICTINDEXOffset** | = 68 *FontDICTINDEXOffset*  The bytes 0C 24 represent the *FontDICTINDEXOffset* operator. The operand byte is hex CF = decimal 207, which is decoded using the rule for byte values from 32 to 246 (see Table 3, Operand Encoding): b0 - 139.This gives the offset of the *FontDICTINDEX*: decimal 68 = hex 44. |
| C3 11 | [ offset ] **CharStringINDEXOffset** | = 56 *CharStringINDEXOffset*  The byte 11 represents the *CharStringINDEXOffset* operator. The operand byte is hex C3, which is the encoded representation of the value 56.This gives the offset of the *CharStringINDEX*: decimal 56 = hex 38. |
| 9B 18 | [ offset ] **VariationStoreOffset** | = 16 *vstore*  The byte 18 represents the *VariationStoreOffset* operator. The operand byte is hex 9B, which is the encoded representation of the value 16.This gives the offset of the *VariationStore* data: decimal 16 = hex 10. |
|  | **GlobalSubrINDEX** | CFF2 offsets: 000C to 000F |
| 00 00 00 00 | count | = empty INDEX; no additional fields represented. |
|  | **VariationStore data** | CFF2 offsets: 0010 to 0037 |
| 00 26 | length | = 38 — length in bytes of the Item Variation Store structure that follows. |
|  | **Item Variation Store** | CFF2 offsets: 0012 to 0037 |
| 00 01 | format | = 1 |
| 00 00 00 0C | variationRegionListOffset | = 12 — offset in bytes from the start of the Item Variation Store. |
| 00 01 | itemVariationDataCount | = 1 — number of ItemVariationData subtables. |
| 00 00 00 1C | itemVariationDataOffsets[0] | = 28 — offset in bytes from start of the ItemVariationStore to ItemVariationData subtable 0. |
|  | **VariationRegionList** | CFF2 offsets: 001E to 002D |
| 00 01 | axisCount | = 1 |
| 00 02 | regionCount | = 2 |
|  | **variationRegions[0]** | CFF2 offsets: 0022 to 0027 |
|  | **regionAxes[0]** | CFF2 offsets: 0022 to 0027 |
| C0 00 | startCoord | = -1.0 (F2DOT14 value) |
| E0 00 | peakCoord | = -0.5 (F2DOT14 value) |
| 00 00 | endCoord | = 0.0 (F2DOT14 value) |
|  | **variationRegions[1]** | CFF2 offsets: 0028 to 002D |
|  | **regionAxes[0]** | CFF2 offsets: 0028 to 002D |
| C0 00 | startCoord | = -1.0 (F2DOT14 value) |
| C0 00 | peakCoord | = -1.0 (F2DOT14 value) |
| E0 00 | endCoord | = -0.5 (F2DOT14 value) |
|  | **ItemVariationData subtable 0** | CFF2 offsets: 002E to 0037 |
| 00 00 | itemCount | = 0 |
| 00 00 | shortDeltaCount | = 0 |
| 00 02 | regionIndexCount | = 2 |
| 00 00 00 01 | regionIndexes[] | = {0, 1} |
|  | **CharStringINDEX** | CFF2 offsets: 0038 to 0043 |
| 00 00 00 02 | count | = 2 |
| 01 | offSize | = 1 |
| 01 03 05 | offsets[] | = {1, 3, 5} (number of elements is count + 1) |
|  | **CharString 0** | CFF2 offsets: 0040 to 0041 |
| 20 0A | [ subr# ] **callsubr** | = -107 *callsubr*  The byte 0A represents the *callsubr* operator. The operand byte is 20, which is the encoded representation of the value -107. |
|  | **CharString 1** | CFF2 offsets: 0042 to 0043 |
| 20 0A | [ subr# ] **callsubr** | = -107 *callsubr*  The byte 0A represents the *callsubr* operator. The operand byte is 20, which is the encoded representation of the value -107. |
|  | **FontDICTINDEX** | CFF2 offsets: 0044 to 004E |
| 00 00 00 01 | count | = 1 |
| 01 | offSize | = 1 |
| 01 05 | offsets[] | = {1, 5} (number of elements is count + 1) |
|  | **FontDICT 0** | CFF2 offsets: 004B to 004E |
| F7 06 DA 12 | [ size offset ] **PrivateDICTOffset** | = 114 79 *PrivateDICTOffset*  The byte 12 represents the *PrivateDICTOffset* operator. The operand bytes F7 06 are the encoded representation of the value 114. The operand byte DA is the encoded representation of the value 79.This gives the size and offset of a PrivateDICT: size is 114 bytes, offset (from start of CFF2 table) is 79 bytes (hex 4F). |
| 77 9F F8 6C 9D AE 9A F4 9A 95 9F B3 9F 8B 8B 8B 8B 85 9A 8B 8B 97 73 8B 8B 8C 80 8B 8B 8B 8D 8B 8B 8C 8A 8B 8B 97 17 06 | [ num\* ] **blend** **BlueValues** | = -20 20 472 18 35 15 105 15 10 20 40 20 0 0 0 0 -6 15 0 0 12 -24 0 0 1 -11 0 0 0 2 0 0 1 -1 0 0 12 *blend* *BlueValues*  The byte 06 represents the *BlueValues* operator. The byte 17 represents the *blend* operator. The operand bytes 77.. 97 are the encoded representation of the values -20… 12. |
| FB 8E 95 86 9D 8B 8B 8D 17 07 | [ num\* ] **blend** **OtherBlues** | = -250 10 -5 18 0 0 2 *blend* *OtherBlues*  The byte 07 represents the *OtherBlues* operator. The byte 17 represents the *blend* operator. The operand bytes FB.. 8D are the encoded representation of the values -250… 2. |
| 77 9F F8 6D 9D AD 9A F3 9A 95 9F B3 9F 08 | [ num\* ] **FamilyBlues** | = -20 20 473 18 34 15 104 15 10 20 40 20 *FamilyBlues*  The byte 08 represents the operator *FamilyBlues*. The operand bytes 77… 9F are the encoded representation of the values -20… 20. |
| FB 8D 95 09 | [ num\* ] **FamilyOtherBlues** | = -249 10 *FamilyOtherBlues*  The byte 09 represents the *FamilyOtherBlues* operator. The operand bytes FB 8D are the encoded representation of the value -249. The operand byte 95 is the encoded representation of the value 10. |
| 1E A0 37 5F 0C 09 | [ num ] **BlueScale** | = 0.0375 *BlueScale*  The bytes 0C 09 represent the *BlueScale* operator. The operand bytes 1E A0 37 5F are the encoded representation of the value 0.0375. |
| 8B 0C 0B | [ num ] **BlueFuzz** | = 0 *BlueFuzz*  The bytes 0C 0B represents the *BlueFuzz* operator. The operand byte 8B is the encoded representation of the value 0. |
| C2 6E 9E 8C 17 0A | [ num\* ] **blend** **StdHW** | = 55 -29 19 1 *blend* *StdHW*  The byte 0A represents the *StdHW* operator. The byte 17 represents the *blend* operator. The operand bytes C2 6E 9E 8C are the encoded representation of the values 55 -29 19 1. |
| DB 57 F7 02 8C 17 0B | [ num\* ] **blend** **StdVW** | = 80 -52 110 1 *blend* *StdVW*  The byte 0B represents the *StdVW* operator. The byte 17 represents the *blend* operator. The operand bytes DB 57 F7 02 8C are the encoded representation of the values 80 -52 110 1. |
| B3 9A 77 9F 82 8A 8D 17 0C 0C | [ num\* ] **blend** **StemSnapH** | = 40 15 -20 20 -9 -1 2 *blend* *StemSnapH*  The bytes 0C 0C represent the *StemSnapH* operator. The byte 17 represents the *blend* operator. The operand bytes B3 9A 77 9F 82 8A 8D are the encoded representation of the values 40 15 -20 20 -9 -1 2. |
| DB 95 57 F7 02 85 8B 8D 17 0C 0D | [ num\* ] **blend** **StemSnapV** | = 80 10 -52 110 -6 0 2 *blend* *StemSnapV*  The bytes 0C 0D represent the *StemSnapV* operator. The byte 17 represents the **blend** operator. The operand bytes DB 95 57 F7 02 85 8B 8D are the encoded representation of the values 80 10 -52 110 -6 0 2. |
| F7 06 13 | [ offset ] **LocalSubrINDEXOffset** | = 114 *Subrs*  The byte 13 represents the *LocalSubrINDEXOffset* operator. The operand bytes F7 06 are the encoded representation of the value 114.This gives the offset (from the start of the PrivateDICT) to the INDEX containing the local subroutines: 114 bytes (hex 72). |
|  | **LocalSubrINDEX** | CFF2 offsets: 00C1 to 00E1 |
| 00 00 00 01 | count | = 1 |
| 01 | offSize | = 1 |
| 01 1B | offsets[] | = {1, 27} (number of elements is count + 1) |
|  | **subr 0** | CFF2 offsets: 00C8 to 00E1 |
| BD BD EF 8C 10 8B 15 | [ num\* ] **blend** num **rmoveto** | = 50 50 100 1 *blend* 0 *rmoveto*  The byte 15 represents the *rmoveto* CharString operator. The byte 10 represents the *blend* CharString operator. The operand bytes BD BD EF 8C are the encoded representation of the values 50 50 100 1. The operand byte 8B is the encoded representation of the value 0. |
| F8 88 07 | [ num ] **vlineto** | = 500 *vlineto*  The byte 07 represents the *vlineto* CharString operator. The operand bytes F8 88 are the encoded representation of the value 500. |
| FC 88 EF F7 5C 8C 10 06 | [ num\* ] **blend** **hlineto** | = -500 100 200 1 *blend* *hlineto*  The byte 06 represents the *hlineto* CharString operator. The byte 10 represents the *blend* CharString operator. The operand bytes FC 88 EF F7 5C 8 C are the encoded representation of the values -500 100 200 1. |