Font/Character Documentation

Greg Nuyens

filed as: {eris}<lispcore>internal>doc>font&chars.tedit
last edited: March 24, 1986

Notice:

this is a draft made available for comments. Please do not forward copies.

Comments are encouraged. Please send them to Nuyens.pa@Xerox.com

---

This note provides information about font and character facilities in Interlisp-D. It is organised in two parts:

1) a user level view of the changes involved in including NS characters in Interlisp-D (adapted from the Koto release notes ({Erinyes}<doc>koto>releasenotes>*))
2) a description of the underlying data-structures and facilities. (exported macros and fns, etc)

User-level view

Interlisp-D now supports the Xerox corporate character code standard, commonly referred to as the NS (Network Systems) encoding, described in the document *Character Code Standard* [Xerox System Integration Standards, XSIS 058404, April 1984]. Previous to the Koto release, Interlisp-D used the ASCII (American Standard Code for Information Interchange) encoding. While the extended-ASCII encoding provided for 8-bit (256 available) characters (primarily Latin alphabet and computer-specific symbols), the NS encoding supports 16-bit (65536 available) characters comprising many foreign alphabets and special symbols.

The benefit of having this large character set, in contrast to approaches that use a small set of character codes and a multiplicity of fonts (e.g., a Greek font, a math font), is that each semantically distinct character is represented by its own character code, completely independent of the character's appearance (font). Thus, the Greek character upper-case Beta is always character code 9794, independent of whether it appears in printed form in a serif style, sans-serif style, italic, etc., and it is unrelated to the Roman letter B (character code 66).

NS characters can be used in strings, litatom print names, symbolic files, or anywhere else that characters can be used. All of the standard string and print name functions (RPLSTRING, GNC, NCHARS, STRPOS, etc.) accept litatoms and strings containing NS characters. For example:

```
_(STRPOS "char" "this is an 8-bit character string")

18

_(STRPOS "char" "celui-ci comporte des charactères NS")

23
```

Characters are organized into 256-member *character sets*, each of which generally consists of semantically related characters. For example, character set 38 is the Greek character set and contains the Greek alphabet and punctuation characters needed to print Greek text. A 16-bit character code thus consists of an 8-bit character set and an 8-bit character number within that set. The ASCII character set is contained in NS character set zero; thus, ASCII characters are still represented by the same 8-bit character codes as previously (i.e., 16-bit character codes whose high 8 bits are zero). Most strings and atoms still consist entirely of characters from character set zero and are represented just as space-efficiently in memory and on files as in earlier releases of Interlisp-D that used only ASCII characters.

In almost all cases, a program does not need to know that it is dealing with 16-bit characters rather than 8-bit characters—the higher level system functions all treat them transparently. The exception is in character-level input/output, where the mportant fact to be aware of is that *characters are not bytes*. The file pointer of a random-access file still counts bytes, and the function NCHARS still counts characters, but the two are no longer directly related. This is discussed in more detail below.

Character-level Input/Output

Incompatible Change: **BIN and BOUT are no longer appropriate for character input/output.**

A character is no longer generally representable in 8 bits. Therefore, characters can no longer, in general, be read or written with the functions BIN and BOUT, which read and write 8-bit quantities. The change is mostly transparent to user programs, especially if those programs use only the higher level functions, such as READ and PRINT. However, it is likely that user programs that manipulated a file character by character using BIN and BOUT should now use the following functions, which may produce or consume more than a single byte:

(READCCODE *STREAM*) [Function]

(PEEKCCODE *STREAM*) [Function]

(PRINTCCODE *CODE STREAM*) [Function]

These functions are documented in the new Interlisp-D Reference Manual. The functions BIN and BOUT are still appropriate for use when reading and writing strictly binary (rather than character) data.

Interlisp-D supports two ways of writing NS characters on files. One way is to write the full 16-bits (two bytes) every time a character is output. The other way, which is the system default, is to use "run-encoding," in which a run of characters in the same character set is written as a sequence of 8-bit character numbers within the character set, preceded by a "change character set" command. The byte 255 (illegal as either a character set number or a character number) followed by a character set number is used to signal a change to a given character set; the following bytes, up until the next change-character set sequence, are all interpreted as coming from the specified character set. Run-encoding can reduce the number of bytes required to encode a string of NS characters, as long as there are long sequences of characters from the same character set, which is usually the case.

Most characters in common use, including those in the ASCII character set, are in character set zero; a file containing only these characters is thus in exactly the same format as in previous releases, viz., one byte per character. However, this should not be relied on.

The fact that the file representation of a character may be more than a single byte has important consequences for any program that uses random access on text files whose characters are run-encoded. First, and most obviously, you cannot count the characters in a string being printed and use that number to derive the file pointer of where the string ends—you must use GETFILEPTR. Second, programs that use SETFILEPTR need to be aware of possible character set changes. At any point when a file is being read or written, it has a "current character set," viz., the character set specified in the most recent "change character set" command written on the file. If the file pointer is changed with SETFILEPTR to a part of the file with a different character set, any characters read or written may have the wrong character set. Programs that use COPYBYTES to copy blocks of characters must ensure both that they are copying on character boundaries and copying to a place that is in the correct character set.

(Internal Note: PRINTCCODE is the user entry to the OUTCHARFN of the stream. It is bounds checked.)

The current character set can be accessed with the following function:

(CHARSET *STREAM CHARACTERSET*) [Function]

Returns the current character set of the stream *STREAM*, or T if *STREAM* is not run-encoded. If *CHARACTERSET* is non-NIL, the current character set for *STREAM* is set. For output streams this causes bytes to be written to the stream if *CHARACTERSET* is different from the current character set; for input streams it merely changes the reader's belief about the current character set. If *CHARACTERSET* is T, run encoding for *STREAM* is disabled—henceforth each character printed to the stream is printed as exactly two bytes (the character set and the character number).

Programs that wish to count characters or avoid worrying about character set changes can thus disable run encoding for a particular stream and count each character as two bytes. There is, however, a cost in file space.

Extensions to CHARCODE

CHARCODE has been extended to allow specifying NS Characters.

CHARCODE has been extended to allow the specification of 16-bit NS characters in multiple character sets. It also uses two new variables, CHARACTERNAMES and CHARACTERSETNAMES, so characters and character sets can be specified symbolically. The new definition is the following:

(CHARCODE *CHAR*) [NLambda Function]

Returns the character code specified by *CHAR* (unevaluated). If *CHAR* is a one-character atom or string, the corresponding character code is simply returned. Thus, (CHARCODE A) is 65, (CHARCODE 0) is 48. If *CHAR* is a multi-character litatom or string, it specifies a character code as described below. If *CHAR* is NIL, CHARCODE simply returns NIL. Finally, if *CHAR* is a list structure, the value is a copy of *CHAR* with all the leaves replaced by the corresponding character codes. For instance, (CHARCODE (A (B C))) => (65 (66 67)).

If a character is specified by a multi-character litatom or string, CHARCODE interprets it as follows:

CR, SPACE, etc. The variable CHARACTERNAMES contains an association list mapping special litatoms to character codes. Among the characters defined this way are CR (13), LF (10), SPACE or SP (32), ESCAPE or ESC (27), BELL (7), BS (8), TAB (9), NULL (0), and DEL (127). Examples: (CHARCODE SPACE) returns 32, and (CHARCODE CR) returns 13.

CHARSET,CHARNUM, CHARSET-CHARNUM If the character specification is a litatom or string of the form CHARSET,CHARNUM or CHARSET-CHARNUM, the character code for the character number CHARNUM in the character set CHARSET is returned. CHARSET is either an octal number, or a litatom in the association list CHARACTERSETNAMES (which defines GREEK, CYRILLIC, etc.). CHARNUM is either an octal number, a single-character litatom, or a litatom from the association list CHARACTERNAMES. Examples: (CHARCODE 12,6), (CHARCODE 12,SPACE), (CHARCODE GREEK,A) and (CHARCODE ^GREEK,A)

Note that if CHARNUM is a single-digit number, it is interpreted as an octal character code, *not* as a character. Thus (CHARCODE GREEK,3) denotes the fourth character in the Greek character set, not the character "3" in that character set.

^CHARSPEC If the character specification is a litatom or string of one of the forms above, preceded by the character "^", this indicates a "control character," derived from the normal character code by clearing the seventh bit (100Q) of the character code (normally set in alphabetic characters). Example: (CHARCODE ^A)

#CHARSPEC (8-bit character codes) If the character specification is a litatom or string of one of the forms above, preceded by the character "#", the eighth bit (200Q), normally zero for 7-bit ASCII characters, is set. This is the way to get character numbers greater than 127. ^ and # can both be set at once. Examples: (CHARCODE #A), (CHARCODE #^GREEK,A)

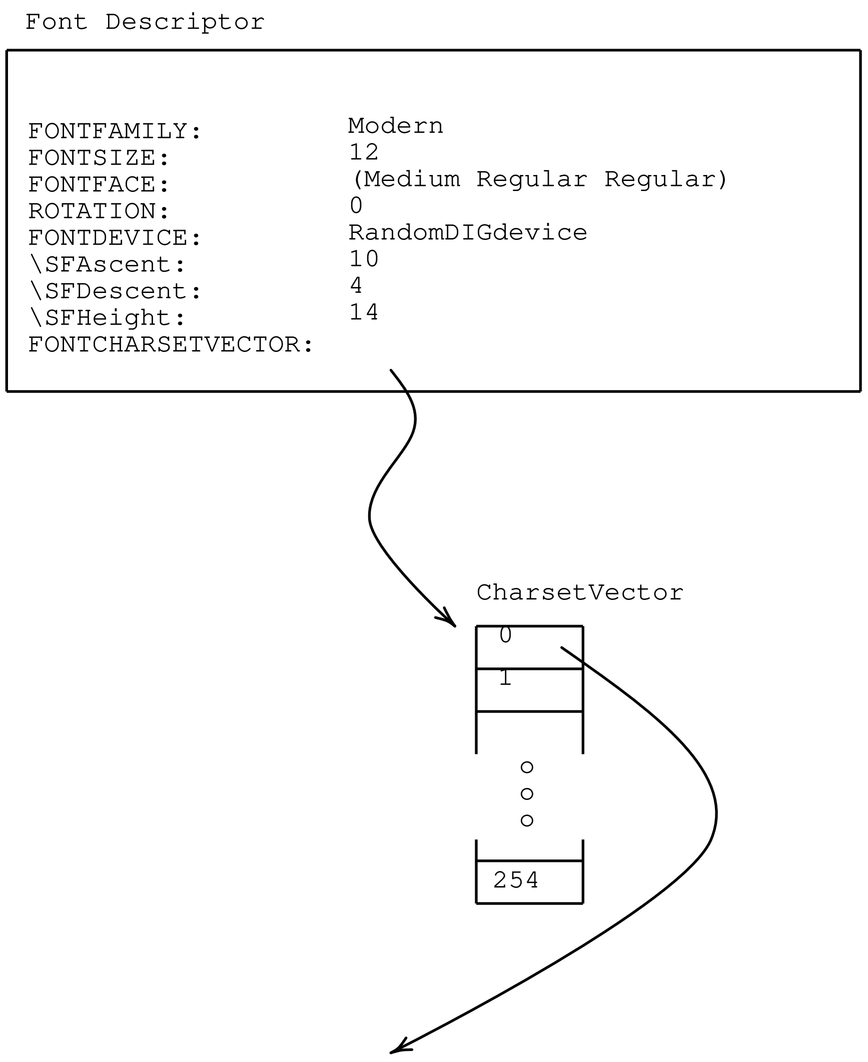
Note: In Intermezzo (and in some other operating systems), characters with the eighth bit set were considered "meta" characters. In the Koto release, however, "meta" means character set 1, and the meta key produces characters with the 400Q bit set, not 200Q.

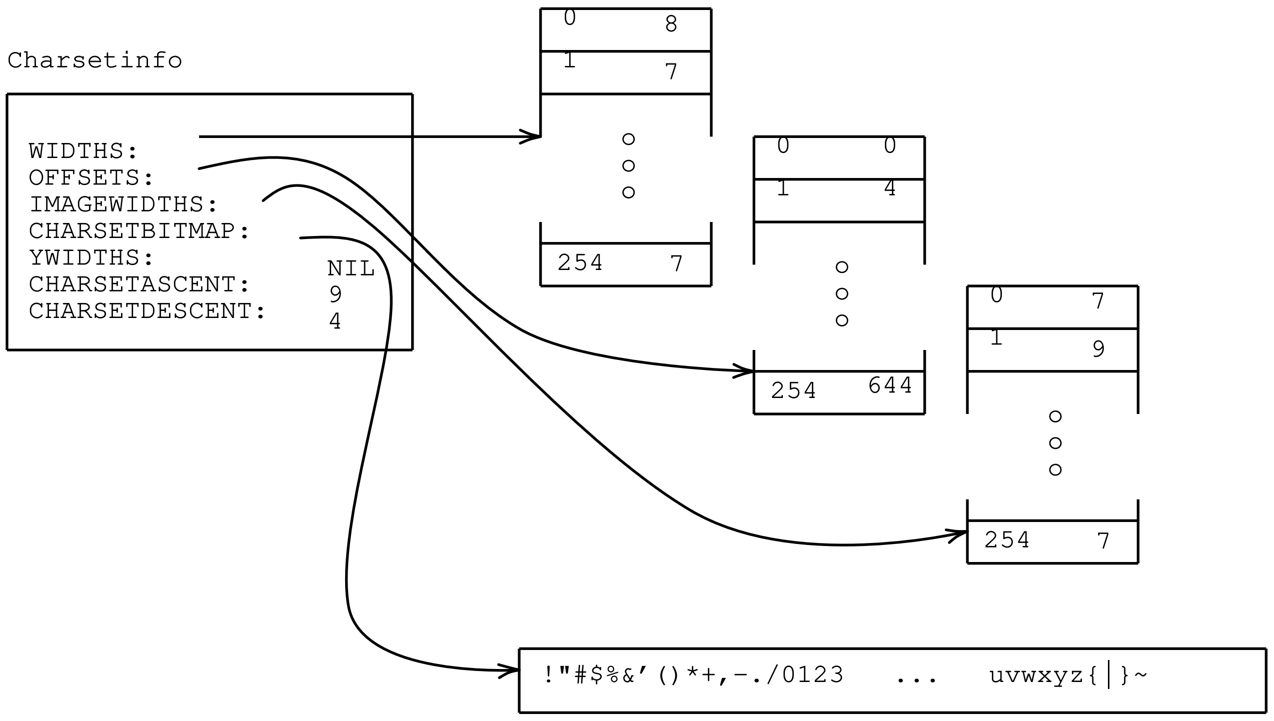
Internals

Note: The information in this section is advisory only. There is no guarantee of back-compatibility in future changes.

Structure of Font Descriptors

To incorporate the increase in information contained in font descriptors due to NS characters, the structure of font descriptors has changed. The current structure is as follows.


Font Descriptor
FONTFAMILY: Modern
FONTSIZE: 12
FONTFACE: (Medium Regular Regular)
ROTATION: 0
FONTDEVICE: RandomDIGdevice
\SFAscent: 10
\SFDescent: 4
\SFHeight: 14
FONTCHARSETVECTOR:
CharsetVector
0
1
254



Description:

<u>Font Descriptors:</u>

The figure represents the logical structure of a font descriptor. Clarification of field values:

Face:	a list containing the weight, slope and expansion. (expansion is always regular)
Rotation:	degrees of rotation
Device:	name of the device for which this font is created
Ascent:	the maximum distance above the baseline for any character in this font.
Descent:	the maximum distance below the baseline for any character in this font. (always positive)
Height:	maximum total height of any character in this font. Equals Descent+Ascent.
FontDeviceSpec:	this is the font specification actually used to create this font after coercions. Thus, if the fontcreate method substituted something other than the original arguments to fontcreate, then this field shows the real contents of the font descriptor, while the family (etc.) fields contain the ostensible (pre-coercion) contents. For instance, display font substitions occur in \CREATECHARSET.DISPLAY according to the variable MISSINGDISPLAYFONTCOERCIONS.
OtherDeviceFontProps:	available to the implementation of each stream. Unexamined by the system.
Charsetvector:	this is a ptr block pointing to the individual charsetinfo's for each of 255 charsets. These are either NIL or a ptr to the charsetinfo. In the diagram, only the charsetinfo for charset 0 is present. When a fontdescriptor is created the charsetvector will already be present.

<u>Charsetinfo:</u>

A charsetinfo contains all the metrics for a single character set (255 characters) of the font.

Widths:	a block of SMALLP's which gives the width in device units for each character (how much to advance the stream xpos after printing this character). This field must be present no matter what the stream type.
Offsets:	integer offsets into the bitmap field, showing the xposition of the beginning of the bitmap for this character. Many streams (especially hardcopy streams) have no need of a bitmap or offsets.
Imagewidths:	This is the width of the image of this character. Often this will be the same as the width, but it can be less or greater. The field must always be valid.
Bitmap:	The bitmap for the characters in this charset. [optional]
YWidths:	currently unused, but for forward compatibility, make this be a block giving the y distance to move for each character (i.e. carriage return should move the y position the height of the font.) [**?** is this a positive distance]
Ascent:	max ascent in the character set.
Descent:	max descent in the character set.

Widths, Offsets, YWidths and Imagewidths are instances of the result of CREATECSINFOELEMENT (see below).

Exported Macros and Functions

(\CHARSET *CHARCODE*)
returns the character set (upper 8 bits) of *CHARCODE*.

(\CHAR8CODE *CHARCODE*)
returns the offset of *CHARCODE* within the character set (the low 8 bits of the character)

(\CREATECHARSET *CHARSET FONT NOSLUG?*)
the createcharset method (determined by the value of the variable IMAGESTREAMTYPES) is called. *NOSLUG?* determines the result if the createcharset method returns NIL. If *NOSLUG?* is NIL, then a "slug" charsetinfo (all characters have a slug (black rectangle) as their image) is returned, otherwise NIL is returned. [This needs improvement, to control how a slug is build. Currently \BUILDSLUGCSINFO is presumed to know how to build a slug for all imagestreams.]

(\GETCHARSETINFO *CHARSET FONTDESC NOSLUG*?)
returns the charsetinfo for *CHARSET* (0..254) from *FONTDESC*. Calls \CREATECHARSET if the charsetinfo wasn't cached already.

(\SETCHARSETINFO *CHARSETVECTOR CHARSET CSINFO*)
will install *CSINFO* as the charsetinfo of (smallp) *CHARSET* in *CHARSETVECTOR*. Since \CREATECHARSET calls \SETCHARSETINFO directly, it usually need not be called.

(\CREATECSINFOELEMENT)
creates a word block for installing as a widths (imagewidths, offsets) field in a csinfo.

(\FGETWIDTH *WIDTHSBLOCK CHAR8CODE*)
returns the smallp width at index *CHAR8CODE*. e.g. (\FGETWIDTH (FETCH (CHARSETINFO WIDTHS) OF csinfo) 55)

(\FSETWIDTH *WIDTHSBLOCK CHAR8CODE WIDTH*)
sets the smallp width at index *CHAR8CODE*.

(\FGETCHARWIDTH *FONTDESC CHARCODE*)
returns the width of any character without having to explicitly fetch the correct charsetinfo for the character set of the character.

(\FGETIMAGEWIDTH *FONT CHARCODE*)
analagous to \FGETWIDTH but for imagewidths (the width of the character image rather than the amount the xposition should be incremented when printing this character.)

(\FGETCHARIMAGEWIDTH *FONT CHARCODE*)
analagous to \FGETCHARWIDTH but for imagewidths.

(\FGETOFFSET *OFFSETBLOCK CHAR8CODE*)
analagous to \FGETWIDTH but for offsets (the position in the bitmap for this character set where the image for this character begins.)

(\FSETOFFSET *OFFSETSBLOCK CHAR8CODE OFFSET*)
sets the smallp offset at index *CHAR8CODE*.