“Unichars”: Unicode Character Repertoire Subsets

IETF 119 ALLDISPATCH

Paul Hoffman/Tim Bray

Using Unicode for the text in your protocols and data formats is good.

Unicode has 1,114,111 code points. Some are problematic. You likely don’t want to allow them.

Unichars discusses what code points are “problematic” and specifies three “Unicode Character Repertoires” that exclude the problematic ones, to varying degrees.

Unichars is based on plenty of input from, and informal rough consensus among, Unicode experts, on the i18ndir@ and art@ mailing lists.

The worst thing that could happen

```
{"example": "\u0000\u0089\uDEAD\uD9BF\uDFFF"}
```

JSON's character repertoire is "All the code points". So this is a perfectly legal JSON text, where the value of "example" is:

1. C0 Control "NUL"
2. C1 Control "CHARACTER TABULATION WITH JUSTIFICATION"
3. Unpaired surrogate U+DEAD
4. Noncharacter U+7FFF (encoded as two surrogates)

Guaranteed to not be useful as text data and is likely to break software. In particular, the "\uDEAD" is likely to generate ill-formed UTF-8.

Problematic: Legacy Control Codes

```
{"example": "\u0000\u0089\uDEAD\uD9BF\uDFFF"}
```

65 code points in the ranges U+0000-U+001F ("C0 Controls") and U+0080-U+009F (“C1 Controls”), plus U+007F, “DEL”.

Left over from pre-Unicode, specifically ISO/IEC 2022 (shudder).

Unichars considers newline (U+000A), carriage return (U+000D), and tab (U+0009) as “useful”. The rest are obsolete and generally lack interoperable semantics, thus problematic. For example, U+0089, CHARACTER TABULATION WITH JUSTIFICATION

Problematic: Surrogates

```
{"example": "\u0000\u0089\uDEAD\uD9BF\uDFFF"}
```

2,048 code points, in the range U+D800 to U+DFFF, used only in the UTF-16 encoding of Unicode and there only in pairs, to identify Unicode code points that don’t fit in 16 bits.

You should use always UTF-8 and never UTF-16 on the wire, so any surrogates are probably symptoms of a bug.

Surrogates can be encoded in UTF-8 but only if you ignore the Unicode spec, which says “Don’t”. Software encountering surrogates has unpredictable and sometimes inconsistent behavior.

Problematic: Noncharacters

```
{"example": "\u0000\u0089\uDEAD\uD9BF\uDFFF"}
```

66 code points:

- The last two entries in each of Unicode's 17 16-bit "planes", U+00FE, U+00FF, U+01FFFE, U+01FFFF, U+02FFFE, U+02FFFF … U+10FFFE, U+10FFFF. They are partly here to support the Byte Order Mark, U+FFEF.
- 32 code points in the range U+FDD0 to U+FDEF. Nobody knows why.

The Unicode spec says these will never be assigned to real characters and should not be used except for "internal" applications.

Character Repertoire: Unicode Scalars

This is all the code points except the surrogates. It is the default repertoire for CBOR and I-JSON.

```
unicode-scalar =
    %x0-D7FF / %xE000-10FFFF  ; exclude surrogates
```

Character Repertoire: XML Characters

Excludes surrogates, legacy C0 Controls, and the noncharacters U+FFFE and U+FFFF. It is the default repertoire for XML, and has had very wide use for many years with few observed character-repertoire problems.

```
xml-character =
   %x9 / %xA / %xD /    ; useful controls
   %x20-D7FF /          ; exclude surrogates
   %xE000-FFFD/         ; exclude FFFE and FFFF nonchars
   %x100000-10FFFF
```

Character Repertoire: Unicode Assignables

All the Unicode code points that are not problematic, i.e. all code points that are currently assigned, or might in future be assigned, to characters that are not legacy control codes.

```
unicode-assignable =
   %x9 / %xA / %xD /                    ; useful controls
   %x20-7E /                            ; exclude C1 Controls and DEL
   %xA0-D7FF /                          ; exclude surrogates
   %xE000-FDCF                          ; exclude FDD0 nonchars
   %xFDF0-FFFD /                        ; exclude FFFE and FFFF nonchars
   %x10000-1FFFD / %x20000-2FFFD /      ; (repeat per plane)
   %x30000-3FFFD / %x40000-4FFFD /
   %x50000-5FFFD / %x60000-6FFFD /
   %x70000-7FFFD / %x80000-8FFFD /
   %x90000-9FFFD / %xA0000-AFFFD /
   %xB0000-BFFFD / %xC0000-CFFFD /
   %xD0000-DFFFD / %xE0000-EFFFD /
   %xF0000-FFFFD / %x100000-10FFFD
```

Relationship to IDNA

If your text field is a domain name, Unichars doesn’t help you. Use IDNA.

If your text field might **contain** a domain name, Unichars “Unicode Assignables” is probably a good choice.

Otherwise, pick the character repertoire that suits your needs.

Relationship to PRECIS

If we were doing PRECIS today, we'd probably start with Unichars "Unicode Assignables".

There's nothing wrong with PRECIS but it doesn't get used as widely as one might expect, possibly because the RFC is a big wall of text.

There is no conflict. It would make perfect sense to say, given a data structure with a name and value, to say that the name is a PRECIS "IdentifierClass" and the value is Unichars "Unicode Assignables".

Draft history

- draft-bray-unichars-00 posted August 2023
- Taken to art@ and i18ndir@ on advice of ART ADs
- 100+ emails, lots of good input, including Unicode guru Asmus Freytag (who is supportive of advancing this document).
- draft-bray-unichars-07 posted October 2023
- Already referenced in draft-ietf-httpbis-sfbis-04 and in discussion of Golang’s next-gen JSON parser