WebAssembly, Unicode and the Web Platform

Or: Concerns about friction on the Web / by non-UTF-8 languages

WebAssembly CG, Daniel Wirtz
Discussion: June 22nd, 2021

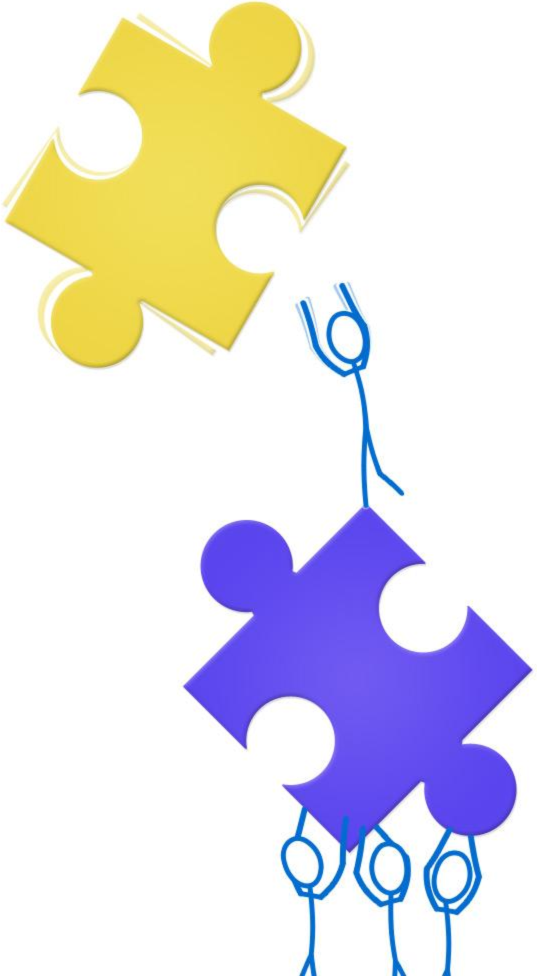
Overview

- **Many languages, incl. JS, utilize WTF-16 string encoding**
  - Overview of relevant Unicode concepts
    - Potentially ill-formed UTF-16 (WTF-16)
    - Surrogate pairs
    - UTF-8
  - Related language concepts
    - Binary strings

- **Proposed canonical ABI desires UTF-8 encoding**
  - Incompatibility of WTF-16 and UTF-8 (WTF-16 ➜ UTF-8 is lossy)
  - Complications when compiling non-UTF-8 languages to Wasm
  - AssemblyScript as an example

- **The Web’s design principle of backwards compatibility**

- **Discussion**
  - What can be done?



Background: Potentially ill-formed UTF-16

“Depending on the programming environment, a Unicode string may or may not be required to be in the corresponding Unicode encoding form. For example, strings in Java, C#, or ECMAScript are Unicode 16-bit strings, but are not necessarily well-formed UTF-16 sequences. In normal processing, it can be far more efficient to allow such strings to contain code unit sequences that are not well-formed UTF-16—that is, [contain] isolated surrogates. Because strings are such a fundamental component of every program, checking for isolated surrogates in every operation that modifies [or here: transfers] strings can create significant overhead, especially because supplementary characters are extremely rare as a percentage of overall text in programs worldwide.”

– **The Unicode Standard, Version 13.0 (§2.7, Unicode Strings)**

Background: WTF-16

"WTF-16 is sometimes used as a shorter name for potentially ill-formed UTF-16, especially in the context of systems [that] were originally designed for UCS-2 and later upgraded to UTF-16 but never enforced well-formedness, either by neglect or because of backward-compatibility constraints." – [The WTF-8 encoding](#) **(§4, Potentially ill-formed UTF-16)**

Backwards compatibility: When switching from WTF-16 to UTF-8, existing string APIs would not match the language's internal string encoding anymore, leaving the language with

i. **changing its string APIs** and breaking all code ever written using these APIs - or -
ii. **keeping its string APIs** at the expense of overhead / complexity.

Affected JS APIs: `charAt`/`charCodeAt`/`codePointAt`, `substring`, `startsWith`/`endsWith`/`includes`/`indexOf`/`lastIndexOf(_, `*`index`*`)`, `split` ➜ `Array#join`, …

Hence: It is unlikely that these languages will be able to change.

Background: Surrogate pairs

“In the UTF-16 encoding form, non-surrogate code points in the range U+0000..U+FFFF are represented as a single 16-bit code unit; code points in the supplementary planes, in the range U+10000..U+10FFFF [more than 16 bits], are represented as pairs of 16-bit code units. These pairs of special code units are known as surrogate pairs.”
– [Unicode Standard](), (§2.5, Encoding Forms)

Example: “𝄞$_{\text{utf16}}$:= 0xD834, 0xDD1E   “𝄞$_{\text{utf16}}$.length := 2

But: “Surrogate code points cannot be conformantly interchanged using Unicode encoding forms. They do not correspond to Unicode scalar values and thus do not have well-formed representations in any Unicode encoding form.”
– **[Unicode Standard]() (§2.4, Code Points and Characters)**

Background: UTF-8

"UTF-8 is a variable-width encoding form, using 8-bit code units, in which the high bits of each code unit indicate the part of the code unit sequence to which each byte belongs."
– **[Unicode Standard](#) (§2.7, Unicode Strings)**

"For UTF-16, most characters can be expressed with one 16-bit code unit, whose value is the same as the code point for the character, but characters with high code point values require a pair of 16-bit surrogate code units instead.

In UTF-8, a character may be expressed with one, two, three, or four bytes, and the relationship between those byte values and the code point value is more complex."
– **[Unicode Standard](#) (§2.5, Encoding Forms)**

Example: “𝄞”_{utf8} := 0xF0, 0x9D, 0x84, 0x9E “𝄞”_{utf8}.length := 4

Note: There are no surrogate code points, so this does not work: $\underbrace{\text{“𝄞”}_{utf16}\text{.substring(1)}}_{\text{now WTF-16}} \nRightarrow \text{???}_{utf8}$

Background: Binary strings

In WTF-16 languages, it is possible to utilize strings as a kind of up-to-16-bit-values immutable buffer for binary data.

- In existing code we want to compile to WebAssembly, and call
  - As with any language-level concept, once the language supports it, people will rely on it
  - Almost certain that we will have to deal with it

- In code interfacing with WebAssembly modules, e.g. JavaScript
  - Used to be common prior to `ArrayBuffer` and its typed views being available
  - Considerable amount of code makes use of binary strings
    - Written before (Chrome 7, Firefox 4, Safari 5.1, IE 10, Opera 12.1, Android 4, iOS 4.2)
    - Targets legacy browsers or restricted/minimal engines
    - Can be used as a technique to minimize code
    - Or out of neglect / appreciation

But: Interpreting a binary string as UTF-8 often corrupts the data, respectively, binary strings may contain values mapping to isolated surrogates when making use of the full 16 bits.

See also: https://developer.mozilla.org/en-US/docs/Web/API/DOMString/Binary

Problem: WTF-16 & UTF-8 are incompatible

“A number of techniques are available for dealing with an isolated surrogate, such as omitting it, converting it into U+FFFD `REPLACEMENT CHARACTER` to produce well-formed UTF-16, or simply halting the processing of the string with an error.“
– **Unicode Standard (§2.7, Unicode Strings)**

Means, converting a WTF-16 string to UTF-8 will sometimes have to either:

i. **Trap (incompatible sequences are rejected)**
   i.e. passing a WTF-16 string over a WebAssembly boundary may (seemingly randomly) trap
   a. Applications may crash mysteriously (say long-running Node.js server applications)
   b. Malicious actors may be able to trigger denial of service intentionally

ii. **Modify the data (incompatible sequences are sanitized, means conversion is lossy)**
   i.e. a WTF-16 string passed over a WebAssembly boundary, may, for instance, not compare equal to the original
   a. Applications may (seemingly randomly) not work/store/read back data properly
   b. Malicious actors may be able to trigger unexpected / undefined behavior
   c. Hypothetical security concerns if authentication tokens, hashes, keys or the likes are affected (Forge, lz-string, window.btoa, …)

Effects may manifest when compiling existing code to WebAssembly, during code migration, etc., either:

i. In code one controls - or -
ii. For any of the same reasons in dependencies (of dependencies) (at some point in the future)

Deciding for a canonical ABI seals the deal

From "Scoping and Layering the Module Linking and Interface Types proposals"

WebAssembly CG
April 27th, 2021

Linked draft PR

Proposed next steps

1. Create a new component-model repo
   a. Containing docs for high-level goals, use case, requirements, FAQ, etc (like the design repo)
   b. Later, merge in the formal spec and spec-interpreter (like the spec repo)
2. Rebase the module-linking repo onto the component-model repo
   a. Use module-linking to initialize the spec+interpreter and continue linking-specific discussions
   b. No core changes are proposed; the "remove duplicate imports?" issue is resolved "no"
3. Rebase the interface-types repo onto the module-linking repo
   a. It's now just a feature proposal, but for the component-model spec
   b. The proposal adds new types and a new definition kind (adapter functions)
4. Split out new adapter-functions repo as a separate feature repo
   a. Adapter functions are the Hard part of Interface Types and there's more churn coming
   b. … but ultimately they are just an *optimization* over using a fixed, canonical ABI
5. Add "canonical adapter functions" to the interface-types proposal
   a. Sidestep hard adapter function design questions by fixing a canonical ABI → Draft is UTF-8 :(
   b. **… allowing module-linking + interface-types to be a component model MVP**

Other notable side-effects and risks

Being limited to UTF-8 at the boundary (to the host or other modules) means:

i. Using non-UTF-8 on both sides would imply re-encoding twice (and potentially trap or modify), that is from non-UTF-8 to UTF-8 and back to non-UTF-8.

ii. …especially in a browser / with JavaScript (WTF-16). Note that some potential optimizations in browsers are neither universal nor standardized.

iii. Risk: It may, theoretically, turn out that an UTF-8-based MVP is good enough for *some* languages, so post-MVP work on the hard parts may slow down, in the worst case leaving problems unaddressed for an extended period of time.

Example: AssemblyScript

Inherits WTF-16 from JavaScript. Worries to be not well served, hence very invested in the topic. Its options are:

- Do nothing and wait/hope for adapter functions or an alternative.
  - Accept double re-encoding overhead and indirect effects on code size (runtime) for its users.
  - Document potential data corruption when using AssemblyScript.
  - Was suggested: Try to use `list u16`, but then cannot talk to WASI and/or browsers directly.

- Switch to UTF-8 and change its string APIs (say, index over bytes).
  - Is a significant undertaking that ultimately breaks with existing AssemblyScript code.
  - Sacrifices more of its goal of being close to JavaScript for JavaScript devs, respectively to compile the same code to both JS and Wasm.

- Provide a separate UTF-8-aware string class.
  - Hurts developer experience, i.e. it's better not having to worry / to work around the problem by hand.
  - Slippery slope of either converting back and forth frequently, or two separate standard library variants for two string classes.

- Work around the problem. (Swift-like breadcrumbs?)
  - Significant work that still hurts performance and increases code size.
  - Similar in nature to “Unistring”*, but with it the engine could at least aid/optimize.

Hence prefers support for WTF-16 in an Interface Types MVP (lift/lower to itself, JavaScript and others), as we currently cannot make any informed decision, also since little is known about what JavaScript and browsers will do in the future.

(*) Can present if there is interest

Design principle: Backwards compatibility

What is a good standard? An essay on W3C's design principles
https://www.w3.org/People/Bos/DesignGuide/compatibility.html

Identifies two kinds of backwards compatibility:

i. New version of a specification with previous version of the same specification
ii. **New technologies with earlier technologies**

“Nobody forgets about the former, because there is nothing the developers of a new version know so well as the previous version they are trying to replace. Backwards compatibility is always hotly discussed.

But the latter is less obvious. It is, in a sense, the complement of extensibility and modularity. Whereas those two stress the importance of developing technology in such a way that it will work together with future new technologies, backwards compatibility stresses the importance of working well with what is already there. No new technology is designed in a void.”

Think: $\text{JavaScript}_{\text{wtf16}} \nRightarrow \text{WebAssembly}_{\text{utf8}}$, with both of them being part of the Web Platform.

Takeaways

- UTF-8 is the more modern encoding, but WTF-16 is the reality we live in.
- Many WTF-16 languages are stuck with their encoding and will likely never change. It's … complicated, but we still want to support them well.
- Interface Types desires only UTF-8 in its MVP, but then is lossy for WTF-16 languages, including when interfacing with JavaScript.
- On the Web, backwards compatibility is important / is a design principle.
- Part of the WebAssembly ecosystem will likely suffer, since legitimizing a canonical ABI with *just one* encoding will produce “winners” and “losers”.

Suggestions: How to produce winners

i. **Spec lift/lower for both a canonical (W/UTF-8) and a well supported legacy (WTF-16) string encoding.**

ii. **Iterate from WTF-16 as the canonical ABI's string encoding** (inverses the "Bringschuld" `DE`).

iii. Not splitting out adapter functions from an Interface Types MVP, so choosing a single canonical encoding becomes unnecessary. Leaves the details open for now.

iv. WTF-8 as the canonical ABI's string encoding to at least tackle the surrogate problem?
   - Does it fit actual languages, or would these have to do a check on the boundary still?
   - Can we reasonably convince ourselves that double re-encoding for WTF-16 languages and JavaScript is OK?
   - What about ill-formed WTF-8 (with surrogate pairs encoded as individual surrogates)?

v. Multiple string types in an MVP, say `string8` and `string16`, and convert implicitly where necessary?

vi. "Unistring"* type that abstracts (Unicode-like) encoding differences away / caches / integrates with IT and GC (same problem).

vii. Other ideas? What do other language implementers think?

Discussion: How can we enable a flourishing polyglot WebAssembly ecosystem, where everyone is a winner? 🎉

(*) Can present if there is interest

Thank you!

… soon may the Wellerman come to bring us sugar and tea and rum “🎶”

a.k.a. "\uD83C" + "\uDFB6"
a.k.a. String.fromCharCode(55356, 57270)
a.k.a. String.fromCodePoint(127926)