

Unicode Handling in BitC[†]

Jonathan Shapiro, Ph.D.
The EROS Group, LLC

May 17, 2010

Abstract

International character set handling basically requires that new programming languages adopt the Unicode standard. There are many choices and options in the standard, and the *model* for how characters and encoding works is quite different from previous models. In consequence, deciding *how* to adopt Unicode is not straightforward.

This note captures both my current summary understanding of how Unicode works and the decisions that have consequently been made in the design of BitC.

1 Introduction

International character set handling basically requires that new programming languages adopt the Unicode [1]. standard. There are many choices and options in the standard, and the *model* for how characters and encoding works is quite different from previous models. In consequence, deciding *how* to adopt Unicode is therefore not straightforward. This document discusses the key concepts in the Unicode standard from the standpoint of the programming language designer.

There are three main issues that any programming language must decide:

1. The encoding and normalization of input units of compilation.
2. What data type will serve as the near-analogue to the `char` type of legacy languages, and incidentally, whether that type will be called "char".
3. The runtime encoding to be used for strings, and the consistency rules to be imposed on string content.

[†] Copyright © 2010, Jonathan S. Shapiro.

THIS SPECIFICATION IS PROVIDED "AS IS" WITHOUT ANY WARRANTIES, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

This note describes how each issue is resolved in BitC, and my rationale.

2 Characters

The conceptual model of characters in Unicode is different from that of ASCII or ISO-Latin-1. I haven't seen a clear exposition of this issue anywhere. Perhaps one exists and I haven't found it. I am merely a Unicode "consumer", not an expert.

A Unicode character is made up of one or more code points. This was done because there were simply too many combinations of glyphs, accents, and modifiers to sensibly encode in any other way. Characters are not uniquely encoded; several *normalizations* are defined. I am uncertain whether a Unicode code point sequence can be decoded into characters successfully without prior knowledge of the normalization.

Key points to remember here:

- The encoding of a unicode character is inherently variable-length.
- Programs that truly wish to handle international character sets *must* use strings, vectors, or some other variable-length representation as the representation for characters.
- In consequence, there is no natural analogy to the legacy notion of a `char` type.

For programmers familiar with ASCII or ISO-Latin-1, these three points are terribly confusing. The confusion is amplified by the fact that, for reasons of backwards compatibility, these older "character sets" have a direct and compatible encoding within the Unicode code point space. When trying to get your head around Unicode, it is best to first ignore these cases, get the big picture straight, and then come back to these. **Code points are not characters.**

Character set processing is complex, and it is best not to have to process characters (in the Unicode sense) within the compiler at all. BitC adopts the following rules to avoid this:

- Input files are encoded in UTF-8 using Normalization C. The tokenizer does not perform any normalization.¹
- String literals are not interpreted by the compiler. Note that this is not a restriction imposed by the language definition.
- Identifiers are encoded according to Unicode conventions, but there is no compiler-level operation that exposes lexical ordering on identifiers. This leaves us free to sort on code points or code units interchangeably for internal purposes.²

Because Unicode characters consist of code point *sequences*, there is no sensible notion of a `char` type corresponding to pre-Unicode programming language notions. Applications that wish to handle characters in the Unicode space in any non-trivial way must deal with the fact that characters are variable length sequences. As a practical consequence, most character processing in such languages must be done using a variable-length representation, such as a string.

3 Code Points

The bulk of the unicode standard deals with the definition of **code points**. The unicode code point space has 21 significant bits, with legal values in the range [0, 0x10FFFF].

Earlier versions of the Unicode standard (Unicode 2.1 and below) had a code point space that could be captured in 16 bits. Unfortunately, many popular languages were designed around this assumption, with the result that choosing a 32-bit representation for code points raises serious interoperability issues. Specifications for these languages are often fuzzy on the distinction between code points and code units, which makes things worse.

¹ There are standardized ways to encode both the encoding and the normalization at the head of a text file. At the moment, I don't know enough about the complexity of doing normalization to admit this into the tokenizer. What I *will* do in the specification is arrange that the meta-encoding for UTF-8/Normalization C will be accepted and ignored if present.

² The problem with normalization in the tokenizer is that identifier [in]equality comparison can fail if you get it wrong.

This problem might turn out to be moot if the unicode code-point classes UCHAR_XID_START and UCHAR_XID_CONTINUE all happen to be non-combining code points, but I haven't had a chance to examine that.

All code points in the Unicode "Basic Multilingual Plane" can be encoded in 16 bits. The extended planes mainly cover synthetic languages, languages that are not seen on computers, and embeddings of legacy encodings like Shift-JIS. It is unclear (to me) how much of a problem it is that (e.g.) Java/C# doesn't readily support the extended planes, but for reasons I will detail below, I suspect the answer is "not much".

For this note, the main points to remember are:

- The `char` type in Java/C# describes a code *unit*, not a code *point*. Specifically, a UCS-2 code unit.
- Even within the Basic Multilingual Plane, there are characters that involve multiple code points, so an instance of the Java/C# `char` type has no meaningful interpretation as a character outside the ASCII and ISO-Latin-1 subsets.

Outside of the ASCII and ISO-Latin-1 subranges of the Unicode space, the type `char` no longer describes a character.

4 Encodings and Code Units

For reasons of space efficiency, there are multiple ways to encode Unicode code points. A given encoding represents code points as **code units**. Code units cannot be interpreted unless the encoding is known.

The Unicode standard specifies encodings into single byte (UCS-1), double byte (UCS-2) and four-byte (UCS-4) encodings. These are, respectively, UTF8, UTF16, and UTF32.

BitC initially chose to make `char` a UCS-4 code unit in order to completely cover the Unicode code point space. This creates major interoperability issues, and given the inherent multi-unit nature of characters it isn't clear that any benefit was had. We will probably shift to a scheme in which UCS1, UCS2 and UCS4 are primary types. While these *could* be encoded as `uint8`, `uint16`, and `uint32`, my intuition is that the code unit type should remain distinguished. If so, then these types must be primary, because the `string` indexing operation must return *something*.

Because of the liberal confusion of legacy `char` type with code units, we are considering *removing* the `char` type in BitC. If we keep it, it will be retained only as a type alias.

Heap vs External There is no need for the heap-encoding of strings to match the external encoding of files.

5 String Encoding

From a language perspective, the Unicode standard raises significant complications about the definition of "string". Is a string:

- A sequence of well-formed characters in *some* (unknown) normalization? This gives maximal freedom to programmers.
- A sequence of well-formed characters in a *particular* normalization? This promotes cross-library consistency.
- A sequence of code points?
- A sequence of code particular code unit?

Since `string` is a primitive type, BitC rejects the first two options, but the latter two are more problematic. The problem is pragmatic: using a UCS-4 encoding in the heap wastes a great deal of space, but any other simple encoding involves a non-unit-time indexing operation.

There is a compromise position, which is where we are currently leaning:

- A well-formed `string` consists of a sequence of code points. The specification does not take a position on the encoding of strings in the heap.
- Strings support indexing on both UCS-2 and UCS-4 code units.
- Any operation that accepts code units and produces a string is obliged to confirm that the code unit sequence constitutes a well-formed code point sequence to ensure that multiple indexing schemes are possible.
- Implementations are encouraged where possible to use a run-encoded internal representation of strings incorporating a hidden cached cursor, such that arbitrary indexing and sequential indexing are both implemented in $O(1)$ time. A reference implementation for such an encoding will eventually be provided by the BitC implementation.

6 Conclusion

So that seems to be where we are at right now. Our understanding of Unicode is still evolving, and we'd appreciate pointers to things we may have gotten wrong.

References

- [1] Unicode Consortium. The Unicode Standard, version 4.1.0, defined by *The Unicode Standard Version 4.0*, Addison Wesley, 2003, ISBN 0-321-18578-1, as amended by *Unicode 4.0.1* and by *Unicode 4.1.0*. <http://www.unicode.org>.