Impulse
Unicode Facilities

Andrew Hunter

August 18, 1999

1 Introduction

It was decided at an early stage to support Unicode in Impulse. Proper internationalisation support is becoming increasingly important with user interfaces today, and Unicode provides a general standard for such support, not just specifying character codes, but also their properties[1]. It is a good choice for a character format, but it is not currently a widely supported storage format, 7-bit ASCII being the only prevalent standard currently, with other 8-bit formats being variously taken up. Impulse will provide facilities to support conversion between Unicode and these formats, to allow maximum portability. This document describes proposals for implementing the standard and the functions that must be associated with it.

1.1 Advantages of the Impulse Unicode implementation

- Portability - the unichar type is guaranteed 16-bit (by virtue of GNU autoconf). wchar_t is too loosely defined to be of any use for anything
- Convienience - no need to interpret the Unicode Character Database yourself
- Portable text formats - support for arbitary 8-bit scripts, UTF-7 & UTF-8 (Unicode itself is not portable, as the standard does not define byte ordering)

2 Important facilities

In order to make sense of Unicode data, we need several facilities.

[1] Unicode is a character-based specification. It does not specify glyph properties, although the Standard makes suggestions as to how a renderer should perform.

2.1 Unicode Character Database

The Unicode Character Database is central to any Unicode implementation. Presently, I use the one provided by Unicode, inc. which is freely available, so updates are easily come by. This comes in the form of a 180k text file, describing the various properties of 6800 characters. The text file is sometimes awkward to work with, and speed is an important consideration when handling text, so it must be translated to a binary form and provided as a data structure.

2.2 Standard character manipulation routines

These include such items as searching, sorting, comparason, etc, and can apply to individual characters or whole strings. Things are complicated by the fact a 'character' can be a sequence of unicode values (Unicode specifies separate diacritics), and ordering can be affected by locale-specific conditions, or made nonsensical. Impulse will do its best to provide a good implementation, and will correct any issues as they arise. As a guideline, we will need analogs of the major string manipulation routines from ANSI C at the very least.

2.3 Normalisation

Any individual Unicode string may be in one of three forms: Unnormalised, Composed (all diacritics contracted to individual codes where possible) and Decomposed (all characters with diacritics expanded to the base character followed by its diacritics in a suitable order). The Unicode Character Database provides decomposed forms of all its characters, so changing a string to decomposed form is quite easy. However, composition is much harder - there are times when ordering of diacritics matter in composing a character, and times when ordering is unimportant (there is a canonical ordering property to help with this problem, though). I plan to use a 'ligature table' consisting of pairs of codes and a suitable composition, which provides high performance, but may not always produce an ideal composition as some characters are decomposed into many codes, with no suitable intermediate stages.

2.4 Regular expressions

There is a technical paper at Unicode on how these should be implemented, which suggests using a Unicode representation of an expression using an analog of current ASCII representations. This fails to realise the benefits of having a massively extended character set - the special regular expression symbols can have their own characters. This removes the need for quoting and potentially increases the readability of regular expressions many times (you can have special glyphs for regexp characters if need be). Regular expression characters break into two groupings - the minimalist set required to specify any regular expression and the extensions that are required to do it efficiently:

2.4.1 Minimalist set

Character code	Symbol	Meaning
U+E000	*	Kleene closure
U+E001	|	Alternatives
U+E002	.	Composition
U+E003	(	Grouping (start)
U+E004	)	Grouping (end)
U+E005	\	Quote

Note that Composition is usually implied by characters being located in a string. It is included for completeness. The quotation character is useful for matching strings which are themselves regular expressions - it should be seldom used itself - maybe it could prove useful for specifying hex character values, too. (Ie QUOTE X <NUMBER>)

2.4.2 Useful extras

Character code	Symbol	Meaning
U+E010	[	Set group begin
U+E011	]	Set group end
U+E012	-	Set range specifier
U+E013	!	Set negation
U+E014	^	String/line beginning meta character
U+E015	$	String end meta character
U+E016	$	Line end meta character
U+E017	$	Paragraph end meta character
U+E018	.	Match anything
U+E019	+	Match 1 or more
U+E01A	?	Match 0 or 1
U+E01B	{	N-ary match specifier begin
U+E01C	}	N-ary match specifier end

Some of these characters are duplicates - I'm trying to use the form you are most likely to be used to as a guide (use the meaning column as definitive). String & Line ends are distinguished between because Unicode has more typographic characters than a mere 'line feed/carriage return' character.

2.5 Locales

Unicode describes a large, and ever-growing range of languages, each of which may have its own semantics. The Impulse Unicode library will provide routines for string translation (similar to GNU gettext, but Unicode) and localisation of functions. Locale configuration files will be written using UTF-8, in order to make them editable in standard text editors.

2.6 UTF-7[2] & UTF-8[3] translations

These two encoding forms provide a way of representing a 16-bit character set using ASCII or an 8-bit encoding. UTF-7 allows Unicode interchange on the Internet[4], and UTF-8 is designed for use in the predominantly 8-bit Unix environment, with particular reference to use in filesystems. Contrary to common sense, neither of these formats is 8-bit (or 7-bit), but actually 16-bit like Unicode. Both provide some compression in favorable circumstances, and have a storage hit in unfavorable ones. These make both make good data interchange schemes, as the byte ordering is well defined in both formats.

2.7 8-bit translations

Prior to Unicode and similar standards, internationalisation of alphabets has been done by defining alternate 8-bit character sets. Microsoft adopted a series of numbered codepages, and ISO defined a set of standards under ISO 8859 (this includes Latin-1). These are all true 8-bit character set representations, with no well-defined method of specifying 16-bit character sets. Impulse Unicode will provide facilities for converting to/from these character sets, but note that there is no sensible manner of dealing with characters outside their domain.

2.8 UTF-16[5]

I would like to support this format too, but I currently have insufficient information on it - any information on sensible conversion strategies would be welcome (does this format define byte ordering?). The Unicode character set is equivalent to this format, and it is a subset of UCS-4 (a 32-bit encoding format)

3 Implementation issues

3.1 Unicode Character Database encoding

This is done using the structures defined in impulse_unidata.h. All the major fields are allowed for, but some are optional, and the descriptional fields (chracter names, unicode 1.0 names, comments) are usually left out to keep source code down. The database is converted into C by unidata2c.pl - this simply maps the descriptions in the Unidata.txt file to their C code equivalents.

3.2 Decomposition

This form of normalisation is easily dealt with, as decomposed forms of all of the characters is maintained in the Unicode Character Database. The decomposition process is thus simply a matter of substituting the sequences there for the

[2] RFC 1642
[3] ISO 10646
[4] The RFC is apparently not yet a standard, though
[5] ISO 10646

character - the diacritic rules should ensure that the decomposition is always correct.

3.3 Composition

This is the hard one. The general idea is to maintain a lookup table of character pairs (2^{32} entries!), with the various possible compositions for those pairs (where more than 2 modifiers may be involved). When a pair is found, a suitable character is substituted, unless the lookup table marks longer sequences, when these are checked. This is likely to be a comparatively slow process.

4 Interface description

4.1 Data types: unichar & unistring

4.2 char *impulse_unicode_to_utf(unistring *string, int bitlength, int options)

4.3 unistring impulse_utf_to_unicode(char *string, int bitlength)

4.4 char *impulse_unicode_to_8bit(unistring *string, impulse_unicode_map_8bit *mapping)

4.5 unistring impulse_8bit_to_unicode(char *string, impulse_unicode_map_8bit *mapping)

4.6 void impulse_unicode_normalise_decompose(unistring *string)

4.7 void impulse_unicode_normalise_compose(unistring *string)