Text Searching Across Multiple Character Sets in Unicode

Philippe Boucher, Bunyip Information Systems Inc., Patrik Fältström, Tele2/SWIPnet,
Jeff Allen, Bunyip Information Systems Inc., Sima Newell, Bunyip Information Systems Inc.

Abstract

Many applications, originally written for 8-bit character sets, are used in contexts where multi-byte character sets would better-support actual data needs. Given the large number of these applications, upgrading them to handle multi-byte character sets without extensive source modifications would be advantageous. Since any such system is not a native Unicode application, data must be processed using specific steps. However, problems relating to string manipulation arise from implicit assumptions made within the existing application. This paper describes one approach to retrofitting multi-byte character set handling into an 8-bit application, discusses what we have learned about the general problem of searching across multiple character sets, and the strengths and weaknesses of using Unicode to address them.

1 Introduction

Although the issue of multiple character set support in Internet-based software is not a new one, much code has already been written that supports only a limited (7 or 8-bit) character set. This leaves the problem of converting this type of software to meet the needs of all Internet users. A prime example is Digger, a Whois++ compliant directory services software system. This application was originally written to use only the ISO-8859-1 character set. Such software is used all over the world by people who speak and write many different languages, and as more people use it, there is a greater need to record information in many native languages using the appropriate scripts. Since Unicode has been proposed as an international character set in developing Internet standards, it is a logical choice to ensure interoperability of the Whois++ directory services system with the greatest number of other applications. Moreover, Unicode encompasses a wide range of characters and its structure allows us to implement more intelligent methods of comparing text strings across multiple languages. One particular difficulty encountered in internationalizing the character set used in a directory services system is performing text-based searches across character sets.

We decided to approach this problem by designing a library of Unicode-handling functions which could be linked into our existing application with minimal modifications to the original source. This paper will describe with the problems that we have encountered in developing a way to make an 8-bit character set implementation Unicode aware. It is the authors' belief that, despite the challenges, other projects could benefit from being made Unicode aware without significant rewrites of existing software.

2 Approach

To build an application that had such up-front requirements for support for searching through multiple character sets, and, in the interest of promoting the use of Unicode in non-Unicode-native applications with minimal code changes, we implemented a C library which can be used to upgrade existing ISO-8859-1 software. This allows applications to be made Unicode aware with a minimum amount of modifications. This section will first describe the methodology which we use in order to process a string when it is first acquired by an application. we will then go on to some of the technical issues and details which had to be dealt with. Our methodology is simple and straightforward, given the constraints of the problem.

2.1 Methodology

A simple process must be followed in order to properly process the string with a minimum of headaches. First, the string is acquired. This is done using whatever code already exists in your application and is independent of the functioning of the library. Once the string is in the application, it may need to be converted if the input character set is known to be something other than Unicode (by default it is now assumed that any input which the application receives would be in the form of a UTF-8 encoded Unicode string). The library provides a conversion function which can transform a generic 8 bit character set to Unicode using dynamically loaded mapping tables. Once the string is converted to Unicode, it must be decomposed and sorted. This is to allow string comparison functions like `strcmp()` to function correctly. The required sequence of events is thus:

Read the translation table(s)
Convert into UCS-2
Decompose according to the Unicode mapping tables
Convert to UTF-8 encoding and return the string

Once we have performed these steps, a string will be in canonical UTF-8 format. It can now be handled normally by a standard C program and strings can be matched using a byte-by-byte comparison such as the `strcmp()` function.

2.1.1 Character set translation

Once an application has been converted to use UTF-8 strings, it will by default assume that its input and output is in Unicode. Although UTF-8 has the property that it cross-compatible with US-ASCII, this is not sufficient. An application should still be able to convert between various 8-bit character sets and Unicode, since some of the
other systems it may have to communicate with might not be Unicode aware and use something other than ASCII. It may also be necessary for the application to come up with some alternate means of representing a Unicode character for some other system which understands only US-ASCII or some other 8-bit character set.

2.1.2 Decomposition

Once the string is acquired and known to be in Unicode, it must be decomposed and its characters sorted according to their canonical ordering priority (COP). This will ensure that when two strings are compared at any point in the application, strings which are the same will match. It also allows for various types of string matching to be possible.

2.2 Implementation

The methodology described above is implemented in the Bunyip Unicode library. Its design is based on the multi-step process required to match text. Text is converted on the way in and out of the library, maximally decomposed internally, and compared using standard methods of analyzing byte-strings. In this section, we discuss some of the finer details of the implementation. The library can be linked into existing source code and provides some simple functions which are used to convert an input string into a canonical UTF-8 Unicode encoding. We chose to use the UTF-8 string representation of Unicode because it is compatible with normal C strings which consist of a byte stream terminated by a NULL (0x00) character.

2.2.1 Character set translation

The library can convert a string using table lookups from an 8-bit character set to Unicode. The data from the translation table can also be used to convert a Unicode string back to the original 8-bit character set. Each 8-bit character in the table is first converted to its Unicode character equivalent with a simple lookup. It is then decomposed, resulting in a string of Unicode characters for the 8-bit character. This string is kept in a hash table. When a Unicode string must be translated back to the original 8-bit character set, lookups are

done in the hash table using the longest possible substring. The result points to an appropriate character from the 8 bit character set.

The same system can also convert any character to some textual string representation. For example, given some characters for which there is no commonly available font, the library can convert these to strings such as `<IMG SRC="somechar.gif">` which could then be sent to a web browser. Note that the proper solution would be to generate the right font for the character in order to display it, but we have found that this ability is often very useful for limited applications on the web.

2.2.2 Decomposition

The library stores the Unicode mapping tables in binary form in a database. Portions of this database can then be dynamically loaded into an application as needed. This way, only the characters which are in use by the application are kept active in memory. Characters which are not used do not take up any resources in the application. The library can use this database to decompose any Unicode character with a single lookup. If the character is not yet present in memory, the block in which the requested character is present will be automatically loaded.

The block of neighbouring characters is loaded, instead of just the character itself, because it is likely that if one character from a particular locale is needed, other characters from the same locale will also be required by the application. For example, if the application requests information about an Arabic character, then it is highly likely that other Arabic characters will be needed as well. Since Unicode tends to group characters from the same locale together, it is more efficient to load a block of characters in one read. This is also good programming practice in general since it minimizes the number of disk accesses.

2.2.3 Matching

For the purposes of our library, we decompose all strings according to the Unicode mapping table and then match them using a byte by byte comparison with the `strcmp()` function. Because we decompose and order the Unicode string, then return it in the UTF-8 representation, methods which compare strings on a byte by byte basis will work perfectly without ever knowing that they are dealing with a Unicode string.

One must keep in mind that, for generic scripts, normal `libc` functions like `strcmp()` on a UTF-8 encoded string are functional, but are perhaps a rather weak definition of a "match". Because of this, we require algorithms which can take care of matching, collation, etc. using locale-dependent functions. Of course, this is true whether or not Unicode is used.

One example of how we define matching in Digger is that `strcmp()` on the UTF-8 encoded string is regarded as an "exact match". The library can also convert UTF-8 Unicode strings to upper case according to the rules in the Unicode mapping tables. This allows us to do case insensitive matching by first converting to upper case and then doing the normal byte by byte character comparison.

We also support something we call "fuzzy" matching which is the use of the Soundex algorithm on UTF-8 encoded strings. Unfortunately, this function is undefined for non-US-ASCII parts of Unicode.

2.2.4 Other Issues

Our library currently allows a pre-existing application to properly use Unicode UTF-8 strings. Internally, the library performs its work using the UCS-2 representation of Unicode. There is full support for going back and forth between UCS-2 and UTF-8. There are also conversion functions which will function with 8-bit character sets which can be translated with lookup tables. The library has been built, and functions well, on 32 bit architectures of both little endian and big endian byte ordering (64 bit architectures are not yet supported). Although the UTF-8 and UCS-2 representation formats are currently supported, it does not currently handle UTF-7 and UCS-4.

There are also some coding issues which still need to be addressed. As it stands, the library implements

functions which were required for the modification of our application. However, these functions are only a subset of those required for a more general library.

One last reminder to all implementers is that we chose to use simple algorithms as opposed to locale-dependent ones in order to have very fast database search abilities. We chose to do this because we are developing what is primarily a database application. The only way to do this with case-insensitive strings is to store all data in the same case. This would mean that there could be only one algorithm used to do the case conversion and this algorithm could not be locale-dependent.

The same is true for any fuzzy searching algorithm used to replace Soundex. The problem with fuzzy matching is that it calculates the distance between the search tokens and all tokens stored in the database according to a locale-specific distance algorithm; then it minimizes this function. This means that the database must be traversed using a linear search, and pre-computed hash values cannot be used. For our application, the time cost was unacceptable, so instead we use Soundex, which can be used to compute hash values. Thus a linear search can be avoided.

3 Discussion

While designing the methodology and the actual code used in the library, we have developed a better understanding of some tricky issues involved in incorporating Unicode into an intrinsically 8-bit application. For example, what happens if a non-Unicode-aware routine naively splits a UTF-8 string? In this part of the paper, we discuss some problems we have encountered and attempt to analyze the overall effectiveness of the library.

3.1 Problems Encountered

We now discuss some of the problems and issues which resulted from our attempts at converting native 8-bit character set applications to Unicode. Many of these could be avoided by writing a native Unicode implementation of our application, but then the point of this exercise would be moot since we are trying to convert existing source with minimum modifications. Some of these problems appear as a direct result of implicit assumptions which come from writing applications which use 8-bit character sets, such as the concept that one character is always one byte. Others come from the added complexity derived from the structure and capabilities of Unicode, such as the fact that characters can be decomposed. These problems can be resolved within the Unicode framework.

3.1.1 UTF-8 String Splitting

Applications will sometimes need to split a string into two or more segments. In an 8-bit character set where a single octet is equivalent to one complete character, this is obviously not a problem. The new string segments are still perfectly valid strings. With a UTF-8 string however, things are not that simple – two problems will arise if such a string is split at an arbitrary position. A UTF-8 encoding of a Unicode character will sometimes require more than a single octet. The first problem is therefore to avoid splitting a string in the middle of a UTF-8 character representation. This is done easily enough by examining the bit pattern of the desired splitting point of the string and working backwards towards the beginning of the string until a proper start of character prefix is encountered. By doing this, we now have complete Unicode characters in both pieces. There is however a second problem. If we have split the string at a character which does not have a canonical ordering priority of 0, the second string segment resulting from this will begin with characters that have no base character to modify. In order to resolve this problem we have to work our way again towards the beginning of the string until we find a complete Unicode character with a COP of 0.

Our library has been implemented with such a function. Given a UTF-8 string and an integer byte length, it will return an offset which indicates where to split the string to avoid both problematic situations while remaining as close as possible to the desired string length.

3.1.2 Tokenization Issues

Some Unicode characters will decompose to a string of characters which may contain some non-modified white spaces. Such a character is U+FDFA (ARABIC LIGATURE SALLALLAHOU ALYHE WASALLAM). Problems arise when a string containing such a character is tokenized. If the string is decomposed before it is tokenized, tokens will be generated from the substrings of characters which were separated by whitespaces. Now, if we search for this character in our set of tokens, we encounter two possible problems. First, if the search character is not decomposed, there will never be any matches. However, if we do decompose the character into its component parts, we can now get partial matches for things which were never requested. Hence, the search results become too generalized. If the string is not decomposed before it is tokenized, we lose some of our ability to do string matching across different locales since we can no longer decompose characters like 'ä' to 'a' followed by '¨'. If we now try to search on a decomposed version of our character, we again get no hits because our set of tokens now contains the original character and not the tokens which are derived from its decomposition. One must note that such characters are not numerous within Unicode, but they do exist, and must therefore be processed in the generic case and not as a specific case.

3.1.3 Decomposition Pitfalls

Applications written using 8-bit character set have certain built in assumptions. A very common one is that given a string, it can be tokenized using the whitespace that it contains (these are space characters, tabs, carriage returns, and linefeeds). With a UTF-8 string this is not necessarily the case as a space character can be modified by a non-spacing mark. The underscore is a perfect example, in ISO-8859-1, it is a single character but in Unicode it decomposes to a space (U+0020) followed by a non spacing underscore (U+0332). If a UTF-8 string containing this structure is tokenized by a normal application, it will split the word in two. What is the correct action in the framework of an 8-bit character set implementation is completely wrong in the context of a Unicode implementation. The right action then gives the incorrect result. A string can be tokenized on characters other than whitespace without the above problem occuring.

There are a number of possible ways to resolve this problem. The first is to properly analyze space characters to see if they are followed by any non-spacing marks. Although this is the correct method for a native Unicode implementation, it requires quite a bit of work in order to convert pre-existing source code to work this way. A second way is to simply not change characters which decompose to a space followed by some non-spacing mark. This way, characters like underscores remain a single byte long in UTF-8 and will be correctly handled by the source code we are trying to convert. This will then permit the modified application to properly tokenize. The drawback to this method is that you must be also able to partially recompose those characters which are merely modified spaces.

Another solution is to provide a tokenization system for an application. This requires, however, that the application not try to do anything based on these problematic characters. We chose to use the second method because it minimizes the changes to the original source. It also has the benefit that, because modified spaces are not decomposed, the application can work with characters such as underscores and circumflexes and still give the expected results.

3.2 String Matching

One of the more important functions of our application is to perform string matching. However, problems occur when a user is trying to search for a name in a language other than his own. For example, consider the case of searching for the name "Fältström". A Swedish user would know to put the umlaut over the 'a' whereas an English one might not. In either case the match must be found, so some method of searching must be used in order to get hits regardless of these locale issues. There are a number of possible approaches. The first is to ignore any non-spacing characters. These are Unicode characters with a non-zero canonical ordering priority. By ignoring these in both the search string and the strings being searched, search hits can be retrieved regardless of whether or not the search string accents were correctly placed. The drawback to this method is that hits may be generated which are actually incorrect.

A second method is to have an equivalence table for characters under a particular locale. Under this scheme, different characters can be mapped to the same character. For example, an 'a' with an umlaut, 'ä' can be mapped back to an 'a'. Therefore, when string comparisons are done, "Faltstrom" will match "Fältström". This method has the benefit that more accurate matching can be done than with the previous, however, the overhead is greater since each locale recognized must have a character equivalence table. This method can be further improved by using tables which encode information regarding the user's locale as well as the locale being searched. However, this requires even more overhead as N^2 tables are required for N locales.

A third method of performing string matching is to use a dynamic alignment algorithm which computes a distance between two strings. This algorithm takes two strings and computes a minimum value representing the difference between them. This value consists of the minimized sum of the characters which have been substituted, deleted, or inserted. The problem with this method is that the distances cannot be precomputed and therefore a linear search must be used.

4 Analysis

We now have a library which, when linked into our application, allows us to use Unicode. The number of changes required in the original source code were minimal. In effect, they amounted to a few function calls to process a string when it is first read, and the modification of a small piece of code where strings were split into two or more segments. The remainder of the code in our application was then able to use UTF-8 strings with no problems.

Although there was some time investment required in order to first create the library, it was no more than the time which would have been required in order to make our application use Unicode natively. However, because we created a system which converts existing source, we can now upgrade some of our other projects with a much smaller investment of time. One must note however, that although there are many applications where this method is useful, it is not a perfect solution which can be applied to every application. There are efficiency issues concerning using this method. It makes an application use UTF-8 strings internally, but it is important to keep in mind that there is some computational overhead in using these strings. They can be much longer than their UCS-2 counterparts, so they can take longer to process and of course require more space. There is also some processing overhead as the library must sometimes convert UTF-8 strings to UCS-2 and then back again to UTF-8 in order to perform certain functions.

5 Conclusions

We feel that this library was successful in achieving our goals. We can now support Unicode in our application and can upgrade some of our other projects to do the same. In its current form, the library is still very specific to the implementation of Digger, however, it has demonstrated that it can allow applications to process Unicode UTF-8 without major rewriting of the source. We plan to extend the library so that it will support UTF-7 and UCS-4 encoding in addition to the current UTF-8 and UCS-2. It will also provide a complete replacement for the `libc` string and character functions such as `strcmp()`, `isalpha()`, etc. in order to further minimize the number of required changes. It will contain systems to assist in tokenization and parsing of Unicode encoded text in its various representations. It will of course be made compliant with the next version of the Unicode character set and standard. There is a lot of software available which can

benefit from this kind of library.

6 Resources

For more information about this Unicode library, see:

http://www.bunyip.com/research/software/unicode/

For more information about the Digger application, see:

http://www.bunyip.com/products/digger/

7 Acknowledgements

The authors would like to acknowledge Leslie Daigle's invaluable assistance in getting this paper completed, as well as Steven Uggowitzer's and Susan Daoud's help for the final editing.

8 References

The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volumes 1 and 2, The Unicode Consortium, Addison-Wesley Publishing, 1990.

Unicode Technical Report #4, The Unicode Standard, Version 1.1, (prepublication edition) The Unicode Consortium, Unicode Inc, 1993.

Architecture of the WHOIS++ Service (RFC 1835), Deutsch, Shoutlz, Faltstrom, and Weider, August 1995.

Architecture of the Whois++ Index Service (RFC 1913), Weider, Fullton, and Spero, February 1996.

The Unicode Consortium Web Server
http://www.unicode.org