Citrus project:
true multilingual support for BSD operating systems

Jun-ichiro itojun Hagino <itojun@iijlab.net>

Research Laboratory, Internet Initiative Japan Inc.
http://citrus.bsdclub.org/index.html

Abstract

Citrus project aims to implement a complete multilingual programming environment for BSD-based operating systems. The goals include:

- ISO C/SUS V2-compatible multilingual programming environment (locale support),
- multi-script framework, which decouples C API and actual external/internal encoding,
- gettext and POSIX NLS catalog,

All of our source code is, and will be distributed under a BSD-like license.

The paper concentrates onto the multi-script framework, which is unique and central to our approach. Most other free software implementations support only Unicode in their multilingual library, or converts external representation into Unicode internal representation and loses significant information on the external representation. We believe a Unicode-only approach is not future-proven, and is not the right way to handle multilingual text. We support multiple different encodings in ISO C/SUS V2 compatible library, and made our library code (as well as user programs) future-proven.

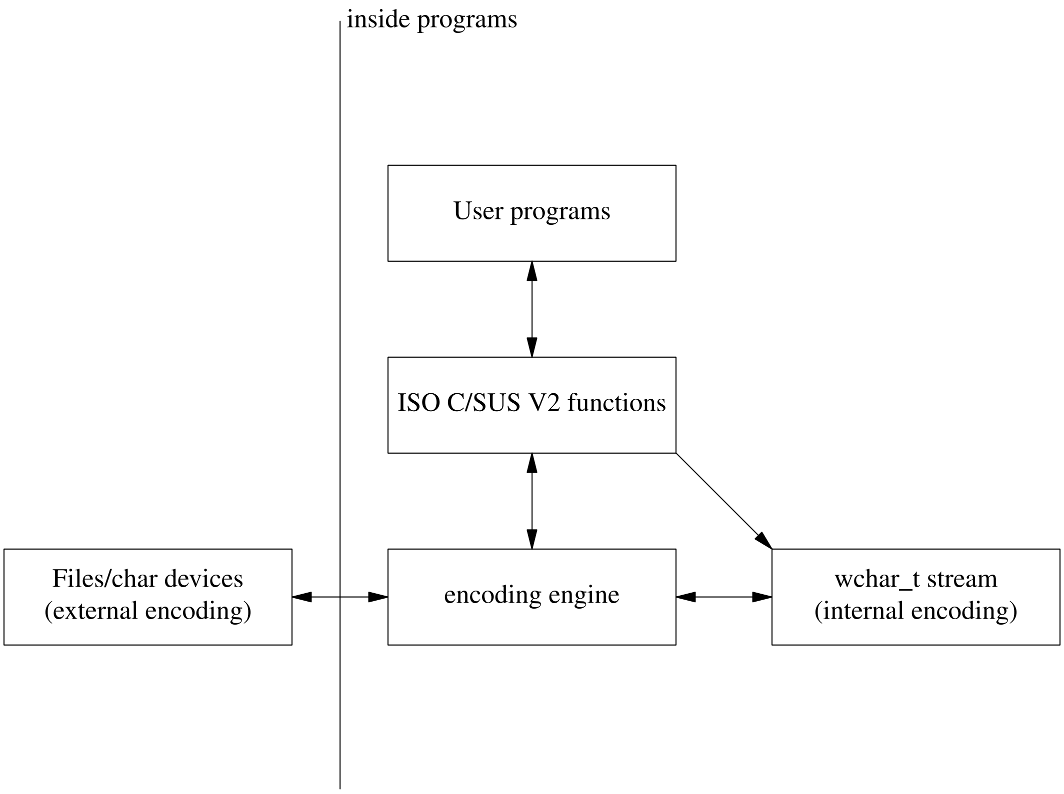
1. Motivation, and multi-script framework

Compared to vendor UN*X operating systems and GNU libc, BSD-based operating systems has been a bit behind in multilingual programming support. This may be because of lack of manpower, difference in interest, or whatever. Anyway, because of the lack of multilingual support, we are seeing increasing pain in porting applications from/to BSD-based operating systems. For example, GNOME, GTK+ and other window manager software relies heavily upon the presence of multilingual libraries. We definitely need a multilingual support library for BSD-based operating systems that is based on ISO C/SUS V2 standards.

The ISO C/SUS V2 standard uses two different encodings for multilingual support: an internal and an external encoding. We use the term "internal encoding" to mean a multilingual encoding system used inside C programs (like inside variables for manipulation), normally declared as an array of wchar_t. On other papers the term "process code" is also used. "External encoding" means a multilingual encoding system used outside of programs (like files or screen output stream). It is also called as "file code". The ISO C/SUS V2 libraries will supply conversion functions between external and internal encoding. Here we call the functions "encoding engine".

Many of existing ISO C/SUS V2 libraries, especially freely available ones like GNU libc, assume that the internal encoding is Unicode-based. More specifically, they assume UCS4 as the internal encoding. They also advocate UTF-8 [Yergeau, 1998] as the external encoding. The encoding engine is hardcoded for internal UCS4 encoding. External encodings other than UTF-8 are supported by conversion functions in encoding engine, which convers external encodings to UCS4, and vice versa. To implement a correct multilingual support, we believe it is very important to not hardcode any encoding, including Unicode. The Citrus project library does not hardcode any existing encoding. In the next section we discuss more fully why Unicode is not enough.

In order to be able to make less assumptions about multilingual encodings, we have designed our libraries to support multiple external encodings, and multiple encoding engines and internal encodings. In other words, each internal encoding has its own encoding engine. We call this concept a "multi-script framework." To support multiple encodings, we simply need to supply the appropriate encoding engines. We also use dynamic loading to load encoding engines, so that we can add more engines on the fly.



2. Why Unicode is not enough

You may be still wondering why we need to support a multi-script framework, instead of hard-coded Unicode support. Below we discuss why Unicode is not sufficient, and why hardcoding of encodings is not preferable.

We specifically have chosen NOT to hard-code Unicode in our library suite. Since we started using computers for text processing, we have experienced many transitions in character set encodings. Here we list our history in chronological order:

- First, we had a total chaos of vendor-defined character set encodings, with 6bit, 7bit or 8bit encodings.
- ASCII-only 7bit encodings (and EBCDIC in some places).
- Hardcoded 8bit encodings. For example, in many of the european countries, ASCII + ISO-8859-1 (so-called Latin-1) has been used. In Japan, JIS X0201, which gives us ASCII variant and Katakana, was widely used.
- Stateful ISO2022 character encodings, with explicit character set designations.

ISO-2022 character encodings allow us to encode multiple language text into a single plaintext stream, and is a good foundation for truely internationalized plaintext handling. For example, by using X11 ctext encoding (which is a subset of ISO-2022 encoding), we can mix Korean, Chinese, Taiwanese, and Japanese text into a single plaintext stream.

Separately from the above direction, there were a couple of regional encodings (encodings that support single language only within a single plaintext stream), including EUC (like euc-kr), MS-Kanji (Shift JIS), or Unicode. Here we list Unicode under "regional encodings", as we cannot mix Chinese/Taiwanese/Japanese text in a plaintext - we need to annotate plaintext with font designation to do it.

Unicode cannot handle multiple Asian characters in a plaintext at the same time, due to "han unification". Unicode maps multiple characters from different Asian regions, into the same Unicode codepoint - this is called "han unification". It was introduced to reduce the amount of codepoint used in Unicode (to fit characters into UCS2 16bit region). While some of the unified chararcters are indeed same across Asian regions, others have totally different glyphs and meanings in different Asian regions [KUBOTA,] . Suppose that we have assigned the same codepoint for O, and O with umlaut. Some people cannot notice the difference, however, this will become a significant problem for people who uses O with umlaut in their language (like for German language). Also note that, if we convert Asian multilingual text into Unicode and then convert it back to other multilingual encoding, the conversion will not be able to preserve the information contained in the original multilingual text, due to the han unification. When we convert the multilingual text into Unicode, we will lose the information encoded in the source due to the han unification.

NOTE: there are proposals to perform language tagging [Whistler, 1999] in Unicode, however, the language tagging jeopardizes one of the very important aspects of Unicode, uniform 32bit wide-char representation, and we do not consider it to be useful.

Every time we change from one encoding system to another, the transition is very painful. In fact, there still are applications that are not 8bit-clean. Even for Unicode, there are implementations assume 16bit UCS2, and they need to migrate to 32bit UCS4. From our experience, it makes no sense any more to pick a single encoding to rely on. We do not advocate the use of ISO-2022, either. We believe that no encodings should be hardcoded, since to do so means that we will have to bear painful transitions over and over again.

Here is another reason for us to avoid hardcoded encoding, and avoid hardcoded Unicode support. There has been widely deployed userbase which uses non-Unicode multilingual text, including big5, euc-kr, KOI8-R, MS-Kanji and some other encodings. If someone says "all people just need to transition to Unicode", that is wrong. Unicode imposes no pain to Latin-1 userbase, while imposing huge pain to Asian and other non-Latin-1 people. And, even if we transition to Unicode, we are unsure if it is going to be enough. So we conclude that we should hardcode no encodings.

What we should do is very similar to the approach with MIME [Freed, 1996] . We will have a way to identify encoding in a plaintext stream, and have support for multiple different encodings. We will switch encoding engines according to the encoding identification. In C library case, we identify encoding by locale settings made by setlocale(3).

3. Gory details

Under our implementation based on multiscript framework, we can switch external encoding, internal encoding and encoding engine as we wish, and we support multiple encodings/engines.

External encoding is normally a stateful, or stateless multibyte encoding, like ISO-2022-JP [Murai, 1993] or UTF-8. An octet stream will be used for multilingual representation inside files. Many of the existing external encodings use variable-length encodings; one letter will be presented as a octet, two octets, or more octets. When we read in external encoding representation into C program, we normally use array of char to hold it. Internal encoding is a stateless, fixed-bitwidth encoding. It is defined internally by the library, depending on the current encoding engine we are using. We use a type called "wchar_t" to hold it. At this moment wchar_t is a 32bit integer (int32_t).

When the external encoding is 8bit (like Latin-1), we can just typecast external encoding (char) into internal encoding (wchar_t). When the external encoding is UTF-8, the natural choice for internal encoding is UCS4. RFC2279 [Yergeau, 1998] defines starndard conversion between them. When the external encoding is ISO-2022, we use a compressed representation of ISO-2022 stream as the internal encoding.

With setlocale(3), we pick a pair of internal and external encoding, and encoding engine. A programmer can convert internal encoding and external encoding, using ISO C/SUS V2-compatible library calls, like mbstowcs(3).

User programs should manipulate wchar_t stream only, and should not manipulate text encoded with external encoding. The details of internal and external encoding are embedded into the library API. Programs should not, and need not to care about internal nor external encoding at all. If a programmer hardcodes some assumption about internal/external encoding to their programs, the programs will not be future-proven. We will also supply wchar_t-ready curses, regex and other libraries, to keep external encoding outside of user programs.

From our strategy, we have two major benefits. First, our library is future-proven, as long as internal encoding fits into the bitwidth of wchar_t (currently 32bit). Next benefit is that we can simplify encoding engine as we wish. Suppose we need to support JIS X0201 or Latin 1 as internal/external encodings. In this case, the encoding engine can be a simple memory copy logic. We do not need to visit a slow encoding engine for simple encodings.

Our strategy avoids problem with Unicode and han unification. Since we can use specific encoding engine that matches the external and internal encodings, we can preserve information supplied by the external data representation, into internal data representation. Therefore, we can have no data lossage during conversion from external to internal encoding, or vice versa.

If we use Unicode as internal and external encoding, we would not be able to enjoy the above mentioned benefits. With UCS4 as the internal encoding, it is very hard to support external encodings other than UTF8. To support them, we would need a huge conversion table to convert the external encoding into the internal encoding, and vice versa. Also, it will not be possible to simplify the econding engine, even if internal/external encodings are simple enough.

4. Multiple encodings

ISO C/SUS V2 multilingual programming API assumes that a single program uses single encoding throughout the program. External and internal encodings are picked when setlocale(3) is called, and the function usually gets called on program startup time. Normally, we cannot determine which encoding should be used, from a wchar_t data stream.

There are various applications that would need a library support for multiple encodings during their runtime session. Examples would be web clients, text editors and email readers. For these applications, we would need to pick internal and external encodings based on input data stream. So, ISO C/SUS V2 API is not sufficient for these type of applications.

We have a temporary workaround to provide a better multiple encodings support. ISO C/SUS V2 API has a data type, mbstate_t, to hold the intermediate state for encoding engine. Our implementation includes a hidden reference to encoding engine inside mbstate_t. By holding an mbstate_t variable with a wchar_t array, we can identify the encoding engine used to encode the wchar_t stream. When we convert wchar_t (internal encoding) back to external encoding, we can automatically use the appropriate encoding engine.

We still are investigating a better API to abstract the manupulation of multiple encodings in a single program. We would like to make a proposal when we are done.

5. Status of implementation

The Citrus project implementation is based on 4.4BSD runelocale implementation [Borman,] . The project expanded it significantly to support multi-script framework, iconv(3), BSD-licensed gettext library, more locale supports like LC_TIME, and expansions for simultaneous multiple locale handling (as presented above). We still are missing multilingual support for stdio routines, like fgetwc function, as it require us to modify FILE structure on each of the BSD system. Depending on the details in standard I/O function implementation, the change to FILE structure may need a libc shared library major number bump. The current implementation supports NetBSD, OpenBSD and FreeBSD. We are trying to finalize our implementation and integrate it into BSDs. NetBSD integration is ahead of other BSDs at this moment. LC_CTYPE locale support and BSD-licensed gettext library were integrated into NetBSD development tree and will be available in NetBSD 1.6. The source code is available via anonymous CVS. The project is definitely an ongoing effort.

At this moment, we are using a tool called mklocale(1) for converting LC_CTYPE locale definition files into binary representation. The mklocale(1) tool and the locale definition file format are derived from runelocale implementation. We should migrate to more standard tool like localedef(1), and the standard file format for locale definition files.

There still are couple of issues to be resolved. We picked 32bit wchar_t for now, just like many of other UNIX operating systems do. It is good enough for future? It is a good question. We believe 32bit is a good compromise. A good thing is that we actually are future-proven. As long as people do not hardcode the current assumption that wchar_t is of 32bit quantity, we will be able to expand widechar representation to 64bit, or something larger. It requires full recompilation of operating system and userland programs, but the transition will impose no change in code. The transition will be just like transitioning from 32bit time_t to 64bit.

For full locale support (including LC_TIME and LC_COLLATE), we will need a large database for localized character tables, time format and others. For voluntary free software effort it can be way too hard to manitain. The maintenance cost for the LC_CTYPE locale databases is also high. We are wondering if we can integrate database from ICU [IBM,] , and reuse it in our multi-script framework.

6. Conclusion

We at Citrus project aim to implement a complete multilingual programming environment for BSD-based operating system. The most important aspect of the effort is multi-script framework. We try to avoid any hardcoded encoding in our library, and embed conversions and encodings into ISO C/SUS V2 API.

DEC/Compaq Tru64Unix uses a similar technology as we have used [Compaq,] , including multiple switchable internal encoding, dynamic library support for additional encodings support, and the use of 32bit wchar_t. As mentioned in the abstract, vendor UNIX implementations are very ahead of free software implementations, regarding to multilingual support. We wish to see more of vendor technologies to be made available to public consumption, under BSD or GNU license.

The author would like to thank Freenix reviewers including Wendy Rannenberg, and Citrus project members including Henry Nelson, for the time they gave me improve the paper.

References

Yergeau, 1998.
F. Yergeau, "UTF-8, a transformation format of ISO 10646" in *RFC2279* (January 1998). ftp://ftp.isi.edu/in-notes/rfc2279.txt.

KUBOTA, .
Tomohiro KUBOTA, *Most Important 1006 Ideographs for Japanese.* http://www.debian.or.jp/~kubota/unicode/.

Whistler, 1999.
K. Whistler and G. Adams, "Language Tagging in Unicode Plain Text" in *RFC2482* (January 1999). ftp://ftp.isi.edu/in-notes/rfc2482.txt.

Freed, 1996.
N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies" in *RFC2045* (November 1996.). ftp://ftp.isi.edu/in-notes/rfc2045.txt.

Murai, 1993.
J. Murai, M. Crispin, and E. van der Poel, "Japanese Character Encoding for Internet Messages" in *RFC1468* (June 1993). ftp://ftp.isi.edu/in-notes/rfc1468.txt.

Borman, .
Paul Borman, *4.4BSD rune(3) implementation.*

IBM, .
IBM, *ICU: International Components for Unicode.* http://oss.software.ibm.com/developerworks/opensource/icu/project/.

Compaq, .
Compaq, "Writing software for the International Market" in *Tru64 UNIX Version 5.1 programming online documentation.* http://tru64unix.compaq.com/faqs/publications/base_doc/DOCUMENTATION/V51_HTML/ARH9YBTE/TITLE.HTM.

Author's address

Jun-ichiro itojun HAGINO
Research Laboratory, Internet Initiative Japan Inc.
Takebashi Yasuda Bldg.,
3-13 Kanda Nishiki-cho,
Chiyoda-ku, Tokyo 101-0054, JAPAN
Tel: +81-3-5259-6350
Fax: +81-3-5259-6351
Email: itojun@iijlab.net