Character Sets in System Libraries

Ulrich Drepper
Cygnus Solutions
1325 Chesapeake Terrace
Sunnyvale, CA 94089

June 24, 1999

1 Introduction

For a long time character sets were not an issue for programming language and library designers. This was because of the lack of different character sets in use in the first place, but also because interoperability between different systems wasn't necessary. Every computer was used in exactly one place and environment.

The first C specification (K&R) and library reflect this situation. There is no support for handling different character sets at the same time at all. The situation changed a bit with the introduction of the ISO C standard in 1990. It defines a first set of functions which can be used to handle character sets. We gracefully skip over the fact that these functions were really useless since at least they tried. A must better and more complete approach comes with the extensions and revisions of the C standard by the amendment 1 and ISO C 9x[1].

In the rest of the document we will describe the functionality available in ISO C 9x and the Unix standard (Unix98) to handle character sets. Beside this we will cover the ISO C++ standard a bit since it introduces its own character set handling functions which are also quite complete and completely different.

2 ISO C Interfaces

For the C interface there are three important standards:

- ISO C 1990 (also ANSI C 1989). This introduced the first, non-reentrant interface for conversion.
- ISO C 1990, amendment 1. This amendment introduces not only a reentrant set of functions for character set conversions. It also introduces standard I/O functions which can take wide character arguments.

[1]Though the 'x' will probably have to be replace by an 'a' since the publication date will slip into 2000.

- ISO C 9x. This standard will finalize the extensions introduced in the amendment 1 and it will also clarify some issues.

2.1 Explicit Conversion Functions

ISO C 1990 defines five functions for character set handling. They can be used to convert between a wide character and a multi-byte character representation of the same character set. Before we can start looking at the actual functions we have to explain what wide and multi-byte character means.

A wide character is an integer value of type `wchar_t`. This type was defined in ISO C 1990. Its exact representation is intentionally not specified. This type is thought to represent character sets which contain more than 256 characters. All wide characters are represented with the same number of bytes. This is what distinguishes this representation from the multi-byte character representation. There is no fixed representation for the latter, there is no special type. Instead multi-byte characters consist of a variable number of `char` values. The number of `chars` (= bytes) depends on the encoding.

The standard does not request a fixed coded character set to be used. In fact, the multi-byte character representation can be used for arbitrary coded character sets. This is the whole purpose as we will see later. But first the functions defined in ISO C 1990:

```
/* Return the length of the multibyte character
   in S, which is no longer than N.  */
extern int mblen (const char *s, size_t n);
/* Return the length of the given multibyte character,
   putting its 'wchar_t' representation in *PWC.  */
extern int mbtowc (wchar_t *restrict pwc,
                   const char *restrict s, size_t n);
/* Put the multibyte character represented
   by WCHAR in S, returning its length.  */
extern int wctomb (char *s, wchar_t wchar);


/* Convert a multibyte string to a wide char string.  */
extern size_t mbstowcs (wchar_t *restrict  pwcs,
                        const char *restrict s, size_t n);
/* Convert a wide char string to multibyte string.  */
extern size_t wcstombs (char *restrict s,
                        const wchar_t *restrict pwcs, size_t n);
```

The `mblen` function returns the number of bytes in the next multi-byte character, given as its first argument. If the value is greater than the second argument, the latter is returned. This function can be implemented using the second function

`mbtowc((wchar_t *) 0, s, n)`

The purpose of `mbtowc` is to convert a multi-byte character into a wide character. The result is stored in the buffer pointed to by the first parameter.

How this transformation is performed is implementation-defined.

The `wctomb` function is performing the conversion in the opposite direction. It takes a wide character and converts it into a multi-byte character (where 'multi' actually can also be one).

Now there is the question how this possibly can be useful. So far we have omitted one detail: the multi-byte character representation (and possibly also the wide character representation) used is determined by the currently selected `LC_CTYPE` category (see `setlocale` in your C manual). The standard only says that the conversion performed is affected by the `LC_CTYPE` selection. Existing implementation are a bit more reliable as they allow the user to specify using `LC_CTYPE` the **external** coded character set, i.e., the representation used in files etc. The internal, wide character encoding is the same for all conversions. This does even allow to convert between different encodings by representing the intermediate result as a wide character string.

To convert entire strings the functions `mbstowcs` and `wcstombs` can be used. They are defined in terms of the functions working on single characters so they do not introduce anything new.

It was initially mentioned that these functions are not useful. This is the result of their handling of stateful character sets. They keep this information internally. Each functions has its own state information from which it start at the next call and there is no possibility to get this information. Therefore we have problems:

- if we try to convert more than one string at the same time (not even in threads, just interrupting the work on the one string for a moment) the state information get messed up. Processing would pick up the state the last call left behind or has to reset the state, which afterwards will lead to problems with the original string since it continues to be processed starting from a wrong state.

- using of the single character and string functions in alteration is not possible since the functions keep their states in different objects.

For this reasons amendment 1 introduces new functions which have an explicit state object in which the implementation has to store the necessary information:

```
/* Determine whether C constitutes a valid (one-byte) multibyte
   character.  */
extern wint_t btowc (int c);

/* Determine whether C corresponds to a member of the extended
   character set whose multibyte representation is a single byte.  */
extern int wctob (wint_t c);

/* Determine whether PS points to an object representing the initial
   state.  */
extern int mbsinit (const mbstate_t *ps);
```

```
/* Write wide character representation of multibyte character pointed
   to by S to PWC.  */
extern size_t mbrtowc (wchar_t *restrict pwc, const char *restrict s,
                       size_t n, mbstate_t *p);

/* Write multibyte representation of wide character WC to S.  */
extern size_t wcrtomb (char *restrict s, wchar_t wc,
                       mbstate_t *restrict ps);

/* Return number of bytes in multibyte character pointed to by S.  */
extern size_t mbrlen (const char *restrict s, size_t n,
                      mbstate_t *restrict ps);

/* Write wide character representation of multibyte character string
   SRC to DST.  */
extern size_t mbsrtowcs (wchar_t *restrict dst,
                         const char **restrict src,
                         size_t len, mbstate_t *restrict ps);

/* Write multibyte character representation of wide character string
   SRC to DST.  */
extern size_t wcsrtombs (char *restrict dst,
                         const wchar_t **restrict src,
                         size_t len, mbstate_t *restrict ps);
```

The important news is the type `mbstate_t`. This is an object which can contain all the information needed to represent conversion states. By passing a pointer to such an object to the 'r' functions above the functions do not depend on any static state local to the functions and so the functions can be called in parallel running threads and also for different inputs.

The remainder of this section contains a brief introduction into using these functions but it certainly does not make reading the full documentation (e.g., in the GNU C library manual) unnecessary.

Files on disk or transferred over network connections etc are almost never encoded using wide character. This would make it hard to exchange data between machines of different endianess. Also, network protocols are usually byte-based which can conflict with a wide character representation (e.g., because of alignments of the receiving buffer).

On the other hand, for normal text processing fixed width character representations are a big win. One does not have to compute the length of each character. Indexing in wide character strings works just like for normal `char` strings. This makes programming much easier and the programs faster. The drawback is that wide character strings often require more space.

If this is accepted there is an obvious need to transform incoming text to wide characters which than can be worked on easily. Afterwards the result has to be transformed back so it can be stored and transferred. This is exactly the problem the functions mentioned above can handle.

An input routines which passes the read data up as a wide character string can be written as follows:

```
wchar_t *
read_wstring (...<something useful>...)
{
  /* Somehow get the data.  */
  char *inbuf = read_data (...);

  /* The state object we will use.  */
  mbstate_t state;
  /* For the result.  */
  size_t len;
  wchar_t *result;

  /* Make sure we start in the initial state.  */
  memset (&state, '\0', sizeof state);
  /* Determine the length of the result.  */
  len = mbsrtowcs (NULL, inbuf, 0, &state);

  /* Allocate room for the result.  */
  result = (wchar_t *) malloc (++len * sizeof wchar_t);

  /* Now the actual conversion.  */
  (void) mbsrtowcs (result, inbuf, len, &state);

  return result;
}
```

This implementation is straight forward. The same function (`mbsrtowcs`) can be used to determine the length of the converted string as well as for performing the conversion itself. So it is possible to allocate enough memory. This two-step process is not always necessary. One can also allocate a buffer and fill it and possibly transform a string in two steps. In this case it might be necessary to keep the object state outside the function and simply pass it in.

Now the tentative reader might ask why it is not necessary to keep the state in the routine above outside at the caller's side. The answer is tricky: since we always transform entire strings including the final NUL byte it is guaranteed that in the end the state objects represent the initial state. This is also why it is not necessary to re-initialize `state` before the second `mbsrtowcs` call. This leads of course to an obvious question: what has to be done to convert an unknown, possibly large amount of text which is not cleanly separated by NUL bytes? It is not possible to convert it by splitting it in individual strings since at the boundaries between the strings, where the NUL bytes to terminate the strings would be inserted, the states would be changed. Some encodings allow such an implementation by splitting at line breaks but it is not generally required.

There are a few alternative approaches for the solution of converting arbitrary much text:

- One can implement a function only using `mbrtowc`. Using this function gives much greater flexibility. The only drawback are the many calls to this function which are necessary.
- Use a pessimistic guess on the number of output characters allowed for the `mbsrtowcs`. I.e., if there is an input buffer with `n` bytes call the function with

  ```
  mbsrtowcs (outbuf, inbuf, n / MB_CUR_LEN, statep)
  ```

  The computation `n / MB_CUR_LEN` makes sure that there are never more input bytes consumed than available. Think about it!
- The system can provide extra functions. This is the case for the GNU C library which provides a much richer set of functionality for this problem.

The conversions in the other direction are very similar and therefore left as an exercise to the reader.

2.2 Implicit conversion

The ISO C standard defines another method for the conversion from multi-byte character text to wide character text and vice versa. It is the logical extension of the well known standard I/O stream interface.

Function likes `printf` now have relatives which can handle wide character strings.

```
/* Write formatted output to stdout.  */
extern int wprintf (const wchar_t *format, ...);

/* Read formatted input from stdin.  */
extern int wscanf (const wchar_t *format, ...);

/* Read a character from STREAM.  */
extern wint_t fgetwc (FILE *stream);

/* Write a character to STREAM.  */
extern wint_t fputwc (wchar_t wc, FILE *stream);
```

These are only a few of the new functions. Almost all existing stream functions now have wide character counterparts. The names are easy to remember, there simply is a 'w' in the name somewhere.

What was said in the last section is also true for these functions. External representations of text is supposed to be in a multi-byte character representation. Therefore the wide character stream output functions are implicitly converting the text they print from the wide character format into a multi-byte format, according to the `LC_CTYPE` category of the currently selected locale. For the `wscanf` and `fgetwc` families of functions the opposite is true, multi-byte external data is converted into wide character data.

Since the conversions are handled all internally in the I/O stream implementation it is an easier interface to convert between multi-byte and wide character representations. Additionally, because of the almost unchanged interface relative to the single byte I/O streams implementation programs can be easily adapted for the handling of larger character sets.

Using the stream I/O functions works quite good as long one does not try to handle more than one external character set at once. If one has to handle more than one, changing the `LC_CTYPE` category during the program runtime is necessary. For each `FILE` descriptor the locale at the time the file was opened is determining the transformation which is performed. I.e., it is possible to switch the locale between different `fopen` calls. But once this is tried in a multi-threaded program it is not safe anymore. For this problem one needs a different interface and this is described in the next section.

3 Unix Conversion Interface

The Unix standard defines another interface to convert character sets. It is by far the most flexible one. It is not tied to a specific locale but instead allows to name the participating coded character sets.

```
/* Allocate descriptor for code conversion from codeset FROMCODE to
   codeset TOCODE.  */
extern iconv_t iconv_open (const char *tocode,
                           const char *fromcode));

/* Convert at most *INBYTESLEFT bytes from *INBUF according to the
   code conversion algorithm specified by CD and place up to
   *OUTBYTESLEFT bytes in buffer at *OUTBUF.  */
extern size_t iconv (iconv_t cd, const char **restrict inbuf,
                     size_t *restrict inbytesleft,
                     char **restrict outbuf,
                     size_t *restrict outbytesleft);

/* Free resources allocated for descriptor CD for code conversion.  */
extern int iconv_close (iconv_t cd);
```

The `iconv_open` function lets the caller name two coded character sets. Of course these must be the coded character set the input buffer is coded in and that the output should use. If such a conversion can be done by the implementation the function returns a descriptor which then can be used in the actual conversion. For this the `iconv` function must be used. It takes position and size of input and output buffer and performs the conversion while copying from the one buffer to the other. By repeated calls arbitrary many text can be converted. Once all the conversion is done the `iconv_close` function should be used to free the resources allocated for the descriptor.

This interface is quite widely available since it is required for Unix compliance. But some vendors did only a half job. They have severe limitation in the functionality. The biggest problem probably is the non-transitivity. If there are conversions $\mathcal{A} \rightarrow \mathcal{B}$ and $\mathcal{B} \rightarrow \mathcal{C}$ available this does *not* mean there is also a conversion $\mathcal{A} \rightarrow \mathcal{C}$ available. And since there is also no way to query the implementation about the available conversions, it is not easily (and portably) possible to find a derivation alone.

The GNU C library implementation does not have this problem. Transitive derivations are always found.

In a later section the techniques used to implement the conversion functions will be explained. At this point it should only be noted that all these operations do not come for free.

4 ISO C++ Interfaces

A third interface is the one defined in the ISO C++ standard, finally finished in 1998. It defines a completely new locale model which is based on templates and is very complex. The locales contain all kinds of information also found in the POSIX locales and much more. The positive thing to say about it is that locales are now actual objects. This means there can be more than one locale at one time in use and multi-threaded programs don't have problems using different locales in different threads.

The character set conversion functionality is also part of the locale, more specific the `codecvt` part. It is a separate object itself and can/must be used to pass information about the coded character sets in use to the functions which need about this. Every stream can have its own locale object, the default is to use the currently selected global locale. Since the character type used in the stream is given as a template parameter adds another level of complexity.

```
template <class charT, class traits = char_traits<charT> >
class basic_ostream : virtual public basic_ios<charT,traits> {
public:
  basic_ostream<charT,traits>& operator<<(bool n);
  basic_ostream<charT,traits>& operator<<(short n);
  basic_ostream<charT,traits>& operator<<(unsigned short n);
  ...
};

template<class charT, class traits>
basic_ostream<charT,traits>& operator<<
  (basic_ostream<charT,traits>&, charT);

template<class charT, class traits>
basic_ostream<char,traits>& operator<<
  (basic_ostream<char,traits>&, char);
```

Only the instantiations with `charT` being `char` or `wchar_t` are pre-defined. They are similar to the C streams. The `char` version does not interpret the

characters and passes the read data directly and unchanged to the user[2]. The wchar_t version performs character set conversions using the codecvt class of the locale of the stream (chosen using imbue or using the global locale.
The interface to the codecvt is very similar to the interface described in the last section.

```
template <class internT, class externT, class stateT>
class codecvt_byname : public codecvt<internT, externT, stateT> {
public:
  codecvt_byname (const char*, size_t refs = 0);
protected:
  ~codecvt_byname ();
  virtual result do_out (stateT& state,
    const internT* from, const internT* from_end,
    const internT*& from_next, externT* to, externT* to_limit,
    externT *& to_next) const;
  virtual result do_in (stateT& state,
    const externT* from, const externT* from_end,
    const externT*& from_next, internT* to, internT* to_limit,
    internT *& to_next) const;
  virtual result do_unshift (stateT& state,
    externT* to, externT* to_limit, externT *& to_next) const;
  virtual int do_encoding () const throw ();
  virtual bool do_always_noconv () const throw ();
  virtual int do_length (const stateT&, const externT* from,
    const externT* end, size_t max) const;
  virtual int do_max_length () const throw ();
};
```

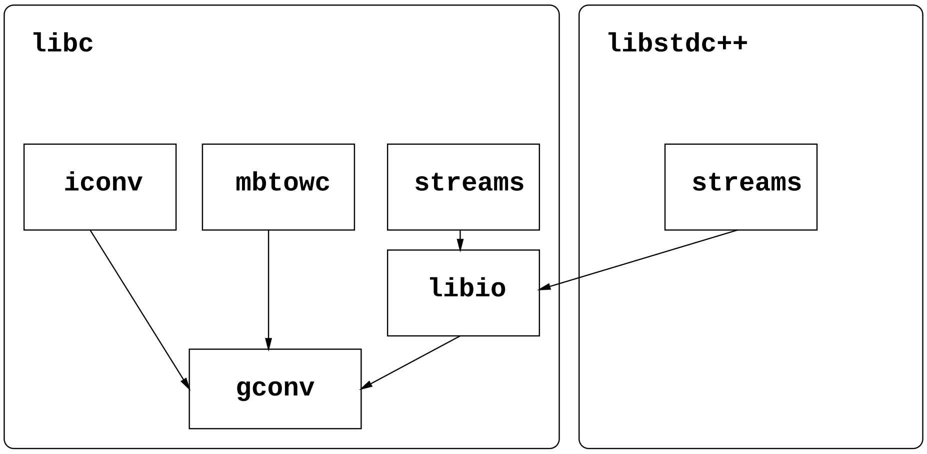
The choice of this interface becomes obvious when one thinks about the tasks to be performed. There is conversion in both direction (for input and output) through do_out and do_in, there is a function which, for stateful encodings, write out code to get into the initial state (do_unshift), a function to determine how much input was necessary to produce certain output (do_length) and there are functions for optimizing (do_encoding and do_always_noconv) and to learn about the size of the output. These are exactly the functions necessary to write an optimized implementation of the I/O streams. To learn more about this interface see the C++ standard.

5 The GNU C library implementation

From the descriptions of the interfaces it should have become clear that the different interfaces are either building on top of each other or are (roughly) equivalent. Therefore the GNU libc implementation does not duplicate all the code with slightly different semantics in several places. Instead it contains a layered implementations where all the different uses depend on the very same,

[2] OK, on some stupid systems newline conversions must be made.

generalized code. This is true even for the C++ part. The `libstdc++` traditionally uses the I/O stream implementation in the C library and therefore it now automatically inherits the character set handling. A schematic explanation of the concept looks like this.



The GNU libc contains a new interface named `gconv`. This is quite similar to `iconv`, but only *quite*. The mapping from `iconv` functionality to `gconv` is trivial.

The implementation of the `mbtowc` family of functions requires a bit more work. The problem is that these functions do not follow the Unix open-use-close model `iconv` is using. In special, there is no place where a descriptor is generated.

One can put the descriptor in the state object passed to the reentrant functions which also is kept internally in the non-reentrant functions. But how is it opened? How are the resources freed?

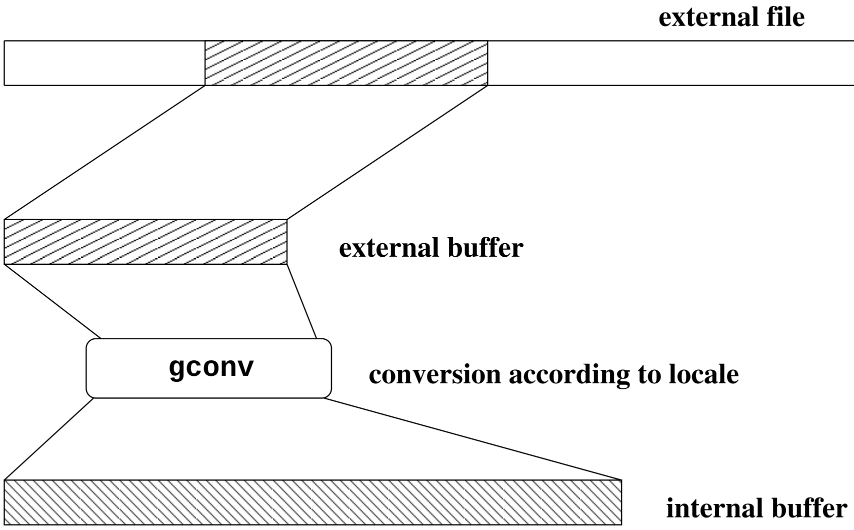
Therefore the implementation goes another route. The descriptors are generated on the fly. But still there is some data to be kept which is local to each conversion. This is then put in the state object which has the type `mbstate_t`. Doing it this way puts a few limitations on the conversions which can be represented but is normally no problem.

The third user of the `gconv` functionality is the low-level part of the I/O stream implementation. It is called `libio` and originally comes from the C++ library. As described before the problem to be solved is to convert external encodings to the internal wide character encoding. The problem is sufficiently interesting and also important to understand (not last for performance reasons) that we devote an extra section to it.

5.1 Wide Character Streams

Uni-byte stream input is relatively simple. When the stream is unbuffered data is read immediately in the user buffer. In case of a buffered stream the data read from the file in a buffer allocated for the file descriptor. The chunks needed by the user are then copied to whatever place s/he wants.

Things are more complicated for wide character streams. The problem is that the external representation is not the same as the internal representation. Therefore it is not possible to simply copy the data from the file into the buffer from which the user receives the data. Instead an intermediate buffer is needed. The intermediate buffer will receive the data from the file. Then in a second step the necessary conversion from the external format to the internal, `wchar_t` format, is performed.



Having this intermediate buffer of course has influences on the performance. Especially certain operations are expensive now.

For starters, the case of a buffered stream should be discussed. The way the implementation of input works if the internal buffer is not empty is the same as for the uni-bytes streams with the exception that the entities are now `wchar_t` (which might, and usually is, larger than a single byte). The question is how do we get the internal buffer filled.

Whenever we have an underflow (i.e., an empty internal buffer) data from the external buffer has to be converted using the `gconv` functions. If the external buffer is also empty, data is simply read from the file. Than the copying conversion can take place which produces the correct text in the internal buffer. As long as the external text is correctly encoded this is quite easy. In case of invalid external data the situation is undefined. But normally this should not happen.

Output works in a very similar way. Text is added to the internal buffer as long as there is room. As soon as the internal buffer is filled the conversion functions are used to convert the data into the external encoding. The result is stored in the external buffer. If the external buffer overflows during the conversion, it is written to disk and then emptied to take the next block of text.

The tricky parts of the whole process are sequence points. At these points in a program the buffer to the next character in the internal buffer must correspond exactly to position of the read/write pointer associated with the file (normally this pointer is in the kernel). For uni-byte streams this can easily be done: one always knows exactly how many more bytes are available (and therefore read) in the input buffer. Therefore a relative seek on the file is sufficient. For output it is simply writing out all data in the buffer which does the job.

For wide character streams the position in the internal buffer does not directly correspond to any position in the file. The conversion is in the way. What has to be found out is exactly what part of the external buffer was necessary to produce the input which was consumed so far. The remaining number of bytes in the input buffer is then the number of bytes the file pointer has to be moved backwards.

If the internal and external coded character set is of fixed-width the calculation is easy. If this is not the case the conversion has to be done again. But this time we stop after the last consumed character in the internal buffer. The position in the input buffer then is the position corresponding to the read pointer in the internal buffer and the file position can be adjusted. Output again is easier as it simply consists of flushing out the text in the internal buffer and making sure it gets written to the file from the external buffer.

Now we are ready to see how input and output using unbuffered streams work. The short answer is: as with buffered I/O with the exception that after every operation there is a sequence point. When unbuffered I/O with uni-byte streams was slow it is *really* slow with wide character streams.
The lessons learned from this are:

- Avoid sequence points whenever possible. Sequence points are calls to `fflush`, `fgetpos`, `fsetpos`, `fseek` and some more.
- Avoid unbuffered wide character streams whenever possible.

6 Details about the `gconv` Implementation

It might be of interest to learn a bit about the techniques used in the implementations of conversion functions. Before we do this a few facts about character sets.

Still in use today are more then 150 character sets (probably far more). Most/All of them can cover the characters contained in the ASCII character set but that is all. There are character set with minor extensions but also drastically different sizes (such as the Taiwanese CNS 11643 standard). Coded character sets might be fixed width and variable width. The ISO 10646 standard can cover most of the character sets and with the possibility of added own extensions it is possible to extend ISO 10646 to cover everything. The available 2^{31} positions provide enough room.

6.1 Different Methods

There are two fundamental methods to implement conversion. The one which can be used for simple conversions is using tables. The same function can be used for all conversions. The only thing changing is the data used. Usually this leads to a small (in code and data size) and fast implementation. The limitation of the table method is obvious, though. Creating a mapping of character values to table entries which works for all coded character sets from a single byte, fixed-width encoding to a variable width encoding with up to 7 bytes is at least difficult. In any way the implementation will be suboptimal for all cases.

Therefore one implements special functions for different encodings. Families of encodings might still be handled by the same function (by using some special tables or by other means) but the result is clear: a specialized implementation is faster. The extreme case is where a single encoding is implemented for each conversion function.

The method for the implementation must be chosen based on the coded character set. There is no general concept.

6.2 Implementation

As a consequence of the sheer number of coded character sets it should be obvious that it is impossible to have all conversion functions in the C library (where the `gconv` implementation has to be). The alternative is to load in the implementations on demand.

For data-driven conversions only the data would have to be loaded if the table lookup function is already loaded. This normally is not the case since there are too many functions required and therefore code as well as data has to be loaded. This means dynamically loadable objects are required.

In the GNU C library this is what is done. Each module implements one or more conversions. When the conversion is needed the ELF object containing the conversion is loaded and it is unloaded after use. Only the system limits the number of modules to be loaded at any one time. Some often used or critical conversions (such as from and to UTF-8) are coded in the C library itself. Their implementation is quite small and therefore does not add much to the overall code size.

The interface of the loadable objects is well documented (which is not the case for the versions of commercial Unices). Therefore it is easily doable to implement own conversions. This might be necessary if the needed conversion is not implemented at all or if it is implemented not as a direct conversion (see the next section about transitivity). What has to be done in addition to writing the module is to add a file of name `gconv-modules` describing the conversions and setting the environment variable `GCONV_PATH` to allow the implementation finding the new files. That is all needed.

6.3 Transitivity

In the section describing `iconv` it was already mentioned that one of the biggest problems of existing implementations is transitivity. The availability of the conversion from $\mathcal{A} \rightarrow \mathcal{B}$ and $\mathcal{B} \rightarrow \mathcal{C}$ does *not* mean there is also a conversion $\mathcal{A} \rightarrow \mathcal{C}$ available. It was also said that the GNU C library implementation does not have this problem. Here we explain how this is possible.

Traditional implementations of the `iconv` function require for their modules (or internal conversion functions) the same interface as for the `iconv` function itself. But this interface does not make it easily possible to execute two or more of the conversions in a row while preserving the interface, side effects and results the `iconv` specification requires.

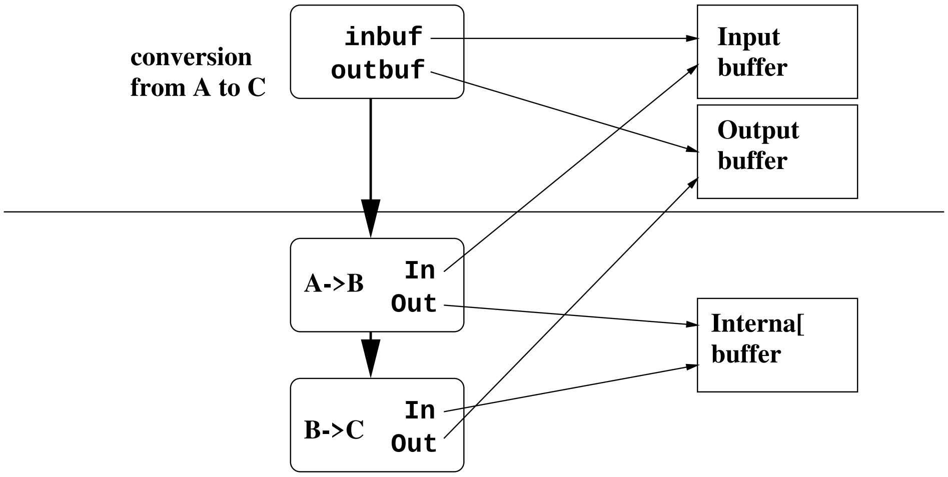
One example should help to see the problem. Given two coded character sets which both use a variable width encoding. When the conversion is performed in two steps there is still the requirement that the pointer to the first unused byte of the input buffer is also the byte which would be used for the next character which will be put out. The problem is that there is an intermediate buffer in the game.

In an ideal implementation the first step would create exactly as much data in the intermediate buffer as required to fill the output buffer. But without performing the conversion itself (both steps) it is impossible to know this number if one or both coded character sets use the variable width encoding.

What has to happen is similar to the flushing of buffers described in the section handling I/O streams. If the second steps stops before reading all bytes from the intermediate buffer the input must be rolled back. It must be determined how much of the input buffer was necessary to produce the text from the intermediate buffer which actually was used. This then lets the implementation determine the pointer to the first unused input bytes.

While this is possible to implement using the existing `iconv` interface it is never done. The consequence is that all commercial Unix implementations which have a `iconv` function do not provide transitivity.

The GNU C library implementation does note have the problem of missing transitivity but it also does not implement the transitivity support the way described above. This would be far too slow since a lot of information which could help to speed up the multi-step implementation is kept inside the `iconv` implementation, without providing the user code access to it. Instead the GNU C library makes the handling of transitivity completely opaque to the user. This way it can use internal information. The used interface must be different from the `iconv` interface which is why the low-level implementation in the GNU libc is called `gconv`.



What the user sees is an interface which requires only one input and one output buffer and in addition the descriptor for the conversion (the `iconv_t` object of the `iconv` interface). The conversion descriptor is the special thing. It contains all the information about the individual steps to be performed. Not only does it reference the modules implementing the single steps of the process, it also contains buffers to the intermediate buffers. The picture above explains how the intermediate buffer gets woven into the conversion. The part above the line is what the user sees, the part below is what actually happens. There is no noticeable difference for the user from the two step process to a single conversion function performing the $\mathcal{A} \rightarrow \mathcal{C}$ conversion. This is the big news about the GNU implementation.

Now it should also become clear what was meant when saying the user might want to replace a slow implementation. Since two step conversions are almost always slower (especially if stateful and/or variable width coded character sets are used) it might be better to have a direct conversion.

7 Conclusion

Character set conversion happens meanwhile in many place, with or without initiative of the user. Switching a program from using uni-byte character streams to wide character streams is on the one hand an easy task since almost all functions for the uni byte interface have a counterpart in the wide character interface.

Where this implicit conversion is not enough explicit conversion routines are available. Not every interface is well designed but when knowing the details and limitations it is possible to use them.

In any way it is necessary to know about the limitation, performance, and complexity of the conversions (implicit or explicit) to write programs which actually can be used. This is important since character set conversion most of the times are performed for human interfaces since computers would not need this in the first place.

A Bibliograpy

There is not much literacy on character set conversion available. Either people keep their knowledge for themselves or the huge amount of information is scaring people. Must-reads include:

- The ISO C standard with the amendment 1 and the draft for the upcoming ISO C 9x standard. All are available from your national standards body.

- The Unix98 specification. It is available online at the website of the Open-Group and can be freely viewed after registration. Or in book form:

  Go Solo 2, The authorized Guide to the Version 2 of the Single UNIX Specification, Prentice Hall. ISBN 0-13-575689-8.

- The Unicode standard. Currently Version 2.1 is the last published version, but only 2.0 is available in printed form:

  The Unicode Standard, Version 2.0, Addison-Wesley Developers Press. ISBN 0-201-48345-9.

- If available, the ISO 10646 standard. It does not add much more information to the Unicode standard and is *extremely* expansive. It is available from your national standards body.

- The GNU C library manual. It contains an entire chapter about character set conversion. This chapter also explains in detail the interfaces of the conversion modules so that people can write their own conversion modules. The manual is distributed in the GNU C library sources.

- *Understanding Japanese Information Processing*, Ken Lunde, O'Reilley & Associates, Inc. ISBN 1-56592-043-0.

  An updated version available also covering Korean and Vietnamese text processing is available.

- One document not at all widely available but nevertheless a good read:

  Manual: Standards for the Electronic Exchange of Personal Data, Part 5: Character Sets. The Hague: Directorate of Departmental Relations and Provisions of Information, Directorate-General for Public Administration, Ministry of the Interior. ISBN 90-5414-019-4.