Information Technology Standards Commission of Japan

ITSCJ

Information Processing Society of Japan
Kikai Shinko Building No. 3-5-8 Shiba-Koen Minato-ku, Tokyo 105, JAPAN

TEL: +81 3 431 2808
FAX: +81 3 431 6493
TX: 02425340 IPSJ J

WG14/N111
X3J11/90-060

To: ISO/IEC JTC1/SC22/WG14 convener
From: Takehisa Inose
Chair IPSJ/ITSCJ/SC22/C WG
Subject: Rationale of Multibyte Support Extension for ISO DIS9899
Date: June/01/1990

Dear Dr. Plauger,

I appreciate your greatly consideration for our proposal of Multibyte Extension for ISO DIS9899.

I attached document "Rationale of Multibyte Support Extension for ISO DIS 9899" .

This rationale summarizes the discussions of C Working Group of SC22 committee of Japan on Draft proposed Multibyte Support Extension of ISO DIS 9899(Document Number WG14/N104). It includes explanations how the working group developed the proposed interface definitions, as well as the concerns on the multibyte character related features in the existing draft ISO DIS9899.

This rationale also includes formal discussions with the X/OPEN internationalization working group as an appendix, so as to keep track the global discussions.

Therefore we believe that this rationale is really helpful to understand the background of our proposed specifications.

We greatly appreciate if you deliver this document to the SC22/WG14 members before the London WG14 meeting so as to help understanding what our proposal is.

Thank you for your cooperation.

Yours very truly,

Takehisa Inose

Enclosures:

Rationale of Multibyte Support Extention for ISO DIS9899

Rationale
of
Multibyte Support Extension
for
DIS9899

DRAFT 1.0

SC22/C WG
IPSJ/ITSCJ
Japan

June 1, 1990

Rationale of Multibyte Support Extension for ISO DIS 9899

ABSTRACT

This rationale summarizes the discussions of C Working Group of SC22 committee, Information Technology Standards Commission of Japan, Information Processing Society of Japan on the Draft Proposed Multibyte Character Support Extension of ISO 9899 [SC22/WG14/N104]. It includes explanations how the working group developed specific interface definitions, as well as the concerns on the multibyte character related features in the existing draft of ISO 9899.

A formal conversation between the working group with X/Open Internationalization Working Group on our previous draft is also appended at the end of this rationale, to keep track the global discussions.

Rationale of Multibyte Support Extension for ISO DIS 9899

CONTENTS

Rationale of Multibyte Support Extension for ISO DIS 9899

1. Introduction

1.1 Purpose

The purpose of this rationale is to clarify and to help understanding on what is defined in *the Draft Proposed Multibyte Support Extension of ISO 9899* [SC22/WG14/N104], by recording the discussions made by *the Working Group* for about one and half year.

1.2 Organization of this Document

Section 1 describes the discussions about generic issues on multibyte character support in *the Standard*. Section 2 covers discussions on the specific interface definitions described in the section 2 of *the Proposed Extension*, while section 3 covers those which do not correspond directly to the sections in *the Proposed Extension*.

Appendix A is the copy of a formal conversation between *the Working Group* and X/Open Internationalization Working Group who is working to define multibyte character support library functions in the future issue of *X/Open Portability Guide*.

Notations

References — All the references are surrounded by brackets []. Wherever it is not described explicitly, references are to *the Standard*.

Multibyte Characters — Multibyte characters in the piece of programs, in the image of output are shown as D1, D2, and so on, while ordinary single byte characters are shown in lower case.

Glossary

- Proposed Extension
- The Working Group
  The sub-working group of SC22/C WG, Information Technology Standards Commission of Japan, Information Processing Society of Japan, which is formed by SC22/C WG to produce *"the Multibyte Support Extension of ANSI C"* and its rationale.
- The Standard
  ANSI C [~~X3J11.158~~-1989] and/or ISO DIS 9899. or ISO/IEC DIS 9899

X3,159

- The Rationale
  The rationale for *Draft Proposed American National Standard for Information System – Programming Language* C [X3J11/88-151].
- IEEE POSIX
- X3J11
- SIGMA Project
- Japanese UNIX System Advisory Committee
  The advisory committee formed by AT&T UNIX Pacific (at that time) with participation of major Japanese companies and universities who were concerned with Japanese capabilities of UNIX System and C language.
  Their proposal regarding the C language extension for Japanese character support using a long char type was submitted not only to AT&T, but also to the *X3J11 committee* through AT&T.

- Japanese C Language Committee
  The voluntary committee of the C language in Japan, who submitted their proposal regarding Japanese character support to the *X3J11 committee*, introducing a letter type for multibyte characters.
  This is *not* a national body of the C standard in Japan. The ITSCJ/SC22/C WG is the only national body corresponding to the ISO/IEC JTC1/SC22/WG14.

1.3 Overview

1.3.1 Policy of the Multibyte Support Extension

Prior to discussions of *the Proposed Extension* in depth, *the Working Group* has discussed its policy of the proposal.

There were two major ideas:

1. To propose some frameworks for extensions of the specification towards multibyte character handling, and then to produce a specification only for Japanese as a practical example and for their own purpose.
2. To propose generic extensions of the specification which can be applied to other languages (that is, other nations and/or cultural regions).

In the light of current trends of international standardization, possible future explosions of markets, and the needs for high portability of application programs, *the Working Group* decided to define extension specification which is generic and applicable to languages of many nations, avoiding to define local specification which depends on Japanese language.

In addition to that, *the Working Group* discussed support of encoding rules. *The Working Group* considered following alternatives:

i. a generic specification to cover all of practical encoding rules that we know.

ii. a limited specification assuming a state-independent encoding in order to avoid related difficulties.

The Working Group considers the the former as its major premise, regarding the policy of *the Proposed Extension* is basically the same with *the Standard*.

With the premise above, *the Working Group* verifies each functional interfaces in *the proposed extension* can be implemented for all of practical encoding rules, precluding all functions which are judged as impractical. Therefore, *the Working Group* believes that all the new functions can be implemented for practical encoding rules, and can be utilized by the application programmers.

However, *the Working Group* feels that specification which is applicable for all theoretically possible encoding rules is just redundant and is impractical, and introduces following assumptions as limitation of encoding rules which is to be supported by new library interface. In other words, the new library interface can be implemented for all encoding rules which obey the limitations:

- Shortest Matching (instantaneous code) – no multibyte character has its byte representation that is the prefix of other multibyte character. This means that in interpreting a byte stream under the encode rule, if a byte sequence matches a certain multibyte character, the multibyte character is determined at that point without checking any following data, because the byte sequence must not be part of any other multibyte character.
- While interpreting a byte data, no two or more multibyte character are determined simultaneously.

These two assumptions are mandatory to guarantee the functionalities of the `fseek()` function and the `ftell()` function for multibyte character handling, even though it is not described explicitly.

The Working Group believes that such limitations on encoding schemes do not exclude any existing code systems from scope of implementations, and so it is practical; it is useful to build clear specification of multibyte/wide character processing.

1.3.2 Scope of the Language Specification Extensions

The Working Group felt that *the Standard* has several points yet to be discussed, as you can see in section 1.4.1, 2.3.7, and 3.4.

From a view point of migration from *the Standard*, *the Working Group* concluded that the language specification should not be modified.

1.3.3 Scope of the Library Functions Extensions

The primitive five multibyte functions in *the Standard* – the `mbtowc()` function, the `wctomb()` function, the `mblen()` function, the `mbstowcs()` function, and the `wcstombs()` function – should be kept as they are, while additional library functions should be defined as a set of higher level functions, assuming the code conversion features of the functions – the `mbtowc()` function and the `wctomb()` function – are available to implement new functions.

Since the five functions cannot process multiple strings simultaneously, these functions may not be used to implement new functions as they are. Section 3.3 and 3.4 discuss about this in details.

For character classification functions and for string handling functions, *the Working Group* proposes another set of functions which perform the same functions as existing `char` functions, but take `wchar_t` objects.

For input/output, *the Working Group* proposes another set of functions which perform input/output of wide character(s) on memory converting from/to multibyte character(s) on external files.

For formatted input/output, *the Working Group* proposes additional conversion specifiers for existing functions (the `printf` family functions and the `scanf` family functions), to convert from multibyte character(s) on the external files to wide character(s) on memory, and vice versa.

1.3.4 On Programming Style

Programs which handle "characters", not only multibyte characters but also single byte characters like ordinary ASCII, can be in unified style and be highly portable by using the new library interface.

The programming style using the new library functions might be as follows; a program reads data onto memory from external files, converting byte sequences to `wchar_t` object using functions like the `getwc()` function, processes `wchar_t` objects just in the same way as ordinary `char` objects, and writes data to external files, converting the `wchar_t` object to byte sequences using functions like the `putwc()` function.

Section 2.1.1 discusses more details about upward compatibility of the new library interface. *The Working Group* imagines that in the near future, the truly internationalized programs must use these new library functions to handle "characters".

1.3.5 On Performance of New Library Functions

The Working Group supposes there exist objections to add new library functions to *the Standard* to support multibyte character processing, with performance concerns, especially from people who mainly need to process single byte codes.

The Working Group believes that implementations are responsible for such possible performance issues, and can avoid them. For example, an implementation may have libraries both for single byte and for multibyte, and application programmers can choose one of these, as appropriate.

The importance of new library functions here is that use of the new interface makes wider and more practical internationalized programming possible, and it could hide the concerns of performance issues.

1.4 General Issues

1.4.1 Type of the wide character

Glossary

- object types [3.1.2.5]
- basic types [3.1.2.5]
- wide string literal [3.1.4],

Concerns

Although this document does not intend to introduce a new definition of the wchar_t, *the Working Group* believes that a type of the wide character would be one of the *object types* and of the *basic types*, just like a long char, as was proposed earlier to the *X3J11 committee*.

This section discusses why *the Working Group* prefers a compiler built-in data type for the wchar_t type.

i. Asymmetry with a char type.

While two expressions for a char type, like signed char and unsigned char, are well defined, the similar expressions for a wchar_t type may not be allowed since they are against the constraints [3.5.2]. Refer to [*the Rationale*, 3.5.6].

A.

```
typedef int wchar_t;
signed wchar_t  wch0;          /* illegal */
unsigned wchar_t wch1;         /* illegal */
```

B.

```
typedef unsigned short wchar_t;
signed wchar_t                 /* illegal */
```

ii. Name space for wchar_t

Since the token wchar_t is not defined as one of *keywords* but as an *identifier* using the typedef specifier, the wchar_t shares a name space with all other *ordinary identifiers*.

For example, a wchar_t identifier cannot be allowed without declaring an appropriate header file <stddef.h> or <stdlib.h>, while a wide character constant L'χ' and a wide character string literal L"χχχ" may always be valid. This implies the specification of the wchar_t is ambiguous and incomplete.

Besides, according to *the Standard*, two identifiers with a lexically identical token may appear in *block scope* [3.1.2.1]. The following shows another abnormal example.

```
#include <stddef.h>

function()
{
  unsigned wchar_t;

  wchar_t = (wchar_t)'T';
}
```

So far, *the Working Group* believes that a type of the wide character would be a compiler built-in basic data type. To support various natural languages in the world, the C programming language must provide a powerful set of functions to handle one of the most important data elements: "characters", which may include multibyte characters, with a simple and unified interface using that data type.

1.4.2 Character sets and Environment

Glossary

- translation environment [2.1.1]
- execution environment [2.1.2]

Concerns

Regarding code conversion for wide character constants, *the Standard* [3.1.3.4] specifies:

> *The value of a wide character constant containing a single multibyte character that maps into a member of the extended execution character set is the wide character (code) corresponding to that multibyte character, as defined by the mbtowc function, with as implementation-defined current locale.*

The Working Group understood that a wide character constant is mapped into the value which corresponds to the execution environment which can be recognized in translation phase. Therefore, when the execution environment is changed dynamically, there may be no portability on the above encoding of wide character constants. For example, the encoding value of a wide character constant `L'D1'` may not be the same as the value by call to the `mbtowc()` function with `'D1'`.

Discussions

This kind of concern, which comes from the difference between translation environment and execution environment, may be found elsewhere like character representation in ASCII/EBCDIC environments and internal representation of floating point, and so forth. Therefore, the issue is not specific to the wide character constants. P Rather the issue should be resolved by an appropriate (static) support of the (cross) compiler, since *the Standard* specifies the actual value of a wide character constant in the execution environments. [3.1.3.4]

Conclusion

The Working Group will propose to include a concern as shown below into the normative addendum to *the Standard*:

> *During an execution of program if the encoding is changed dynamically with the* `setlocale()` *function, the behavior of code conversions, including those for character constants or for string literals, is undefined.*

2. Rationale - corresponding to SC22/WG14/N104

2.1 Environment and Language [2]

2.1.1 Upward Compatibility of Text Handling Programs

Concerns

Some people might be anxious about the way of text processing using wide character. They might fear that the new type `wchar_t` (though it really is a `typedef` type) would force them to give away their well-established programming style of text processing.

Discussions

Wide character was introduced in *the Standard* in order to regard any text data as a sequence of logical 'character' (element of extended character set). If we use the wide character, we need not to suffer from bothersome byte handlings for multibyte characters. In fact, it avoids tremendous operations to handle the multibyte character encoding (for example, detecting boundary between two multibyte characters, and keeping the shift-state if there is on a shift-encoding).

It seems quite clear that designers of the C programming language and most of the C users believed an assumption that they could express any single character with one byte. Note that the C programming style of text processing had been established under that assumption. This means that such programs would manipulate each byte object as a character.

However, this assumption is no longer true in text processing when the text data may include multibyte characters. If we try to write some programs to handle multibyte characters in byte representation, we will face with several problems as follows:

- It is necessary to interpret shift-sequences and to determine the shift-state for any text streams.
- It is very difficult to split and/or to concatenate character strings.
- It is almost impossible to ''seek'' file position indicator in the streams.

No doubt, programs developed under the byte-oriented text processing style, cannot work well in the multibyte character environment.

In order to promote portability of programs between the single byte character environments and the multibyte character environments, it is necessary to solve the issues. As hinted above, it is a wide character that provides a good solution for the problems, hiding unnecessary difficulties about a multibyte character encoding from the user and thus keeping the well-established programming style as well.

To achieve this, a full set of wide character functions should be provided in *the Standard*. Since such a full set in *the Proposed Extension* is going to include the functions each of which is corresponding to one in the traditional libraries, including character handling `<ctype.h>`, string handling `<string.h>`, and input/output `<stdio.h>` functions, it will be possible with ease to convert most of the programs with the traditional programming style for the single byte character environment, into what is suitable for both the single byte character environment and the multibyte character environment; the wide character programming style.

Conclusion

The Proposed Extension introduces the full set of wide character functions with maximum upward compatibility for single byte character processing, which enables the traditional text handling programs to be still useful for both the single byte and multibyte character environments with minimum adaptation or modification.

For example, a text handling program for the traditional environment (that is, `char == 8 bit`) will be useful if we rewrite it in the following way:

char	☞	wchar_t
strcat	☞	wcscat
fgets	☞	fgetws

We will expect that the new program will be executed in almost the same performance as previous version under the following environment:

`wchar_t == 8 bit`
ASCII,
non-shift encoding, no code conversion

and that the new program will be useful in the following multibyte character encoding:

`wchar_t == 16 bit`
ASCII + JIS X 0208 (Japanese characters),
shift-encoding, with code conversion.

2.1.2 Wide characters and shift-sequence characters

Concerns

Among the topics in our discussion, there are:

i. what a shift-encoding is (the definition of shift-encoding), and

ii. how we treat a shift-sequence.

The Working Group strongly feels that *the Standard* doesn't specify shift-sequence clearly enough to image some realistic implementation, and that it is necessary to clarify what a shift-sequence is in *the Proposed Extension*.

Discussions

In most encodings, a shift-sequence can be regarded not as a character that is meaningful for human but as what is necessary to express text, consisting of a large number of different characters from among several character sets, on a byte data stream.

The Working Group regards a shift-sequence as a character:

- that is not meaningful for human, and
- that specifies the new shift-state applied for the following byte interpretation, regardless the previous state.

It is not desirable that it appears in wide character strings because of its meaningless not only for human, but also for wide character processing.

While *the Standard* has already defined a shift-sequence is one of the multibyte characters, there is no clear definition about whether or not the shift-sequence is a legal component of wide characters.

The Working Group concentrates on the discussion about encoding rules where a shift-sequence is a multibyte character, but is not a wide character. Also, *the Working Group* considers the feasible implementations under such encodings.

We may think of some wide character sequence that consists of the corresponding wide characters to members of a multibyte character. For example, we may think of a wide character sequence `L"\033$B"`, which consists of three wide characters, `L'\033'`,`L'$'`,`L'B'`, corresponding to a three octet (byte) escape sequence for designation of JIS X 0208-1983, `ESC 2/4 4/2`.

The Working Group cannot find any positive necessity to allow such wide character sequences. Different implementations may convert it to different results. In some implementations, a code conversion function invoked with such wide character string might well go mad. Thus we conclude that such kind of wide character sequences be invalid because no such wide character sequences should appear in Input/Output text data stream of type wchar_t.

Conclusion

A shift-sequence is a multibyte character, not a wide character.

The result data being converted from a wide character string that consists of the abnormal wide characters corresponding to bytes of a certain multibyte character is undefined.

2.1.3 Constraint for the execution character set

Concerns

The Standard describes few about encoding rules. This lack of specification lead us to have no way to reject the following *unusual* encoding rule, which we would not intend to accept or use.

Discussions

Let us consider the following encoding rule:

- The character set is union of ASCII and the following three characters; 'O', '□', and '∇'.
- All single bytes except for 0x40 and 0x24 are treated in the same way as in ASCII.
- Any single 0x24 (followed any byte except for 0x40) represents single character 'O'.
- If A byte 0x24 precedes 0x40, both two bytes represent single character '□'.
- Two adjacent 0x40's mean '∇'.
- It is error if there are any byte 0x40 following one except for 0x24.

One way to interpret the above encoding is as follows; Any bytes except for 0x24 and 0x40 are mapped to ASCII character set as soon as it appears. A byte 0x24 preceding one except for 0x40 can be treated as 'O'. Apart from these two trivial cases, it is tremendous to interpret byte sequence which consists of a single byte 0x24 followed by two or more 0x40s. In order to interpret this type of sequence, we would count the number of 0x40's. If the number were even, the result would be a 'O' followed by appropriate number of '∇'s. Otherwise if it were odd, a '□' and subsequent '∇'s we would get.

It would be much difficult to implement a code conversion function under this encoding because if it had had found 0x40 following single 0x24 it should have scanned the byte sequence until it would meet one but 0x40. This would have caused the implementation to hold an un-predefined size of, that would be infinite size, of buffer. It is clear that this requirement is not realistic at all.

According to the description in *the Standard*, there seems to be no way to reject such an terrible encoding. By the way, *the Standard* defines a constant MB_CUR_MAX, which specifies maximum number of bytes as a result of the wctomb() function. It is natural to regard the constant as the size of internal buffer for code-conversion. It implies that *the Standard* never allows such an encoding rule that there is some possibility to read infinite number of bytes in order to get a character during code conversion. Thus we decide that it is feasible to impose the existence of finite number of byte pre-reading on the encoding rule.

There should be such property of encoding rules that we can easily determine boundary between two multibyte character on a byte stream. This property is necessary to ensure the functionalities of the fseek() function [4.9.9.2] and the fgetpos() function [4.9.9.1] on multibyte character streams. Thus we should deny such encoding rules that; two or more characters may be simultaneously found out from a byte stream.

Conclusion

In order to ensure the following requirements:

- the maximum number of bytes need to pre-read in getting single character is finite and determinable, and
- at most only one character is returned once at a byte feeding to the code conversion function

in any encoding rules according to it, we decide to introduce the following specification:

Any encoding rules should be instantaneous, where shortest-matching interpretation is available.

Shortest-Matching means that if you find a byte sequence which corresponds to a certain multibyte character at a point, you need not to think of any subsequent bytes to establish the multibyte character. Both this specification and the existence of multibyte character ensure that the encoding rules be in the above requirements.

2.1.4 Execution character set and the setlocale() function

Concerns

The current value of the locale category LC_CTYPE determines the encoding rule which the code conversion functions obey. This means an invocation of the `setlocale()` function which will change LC_CTYPE category causes the code conversion functions to change its encoding rule.

Switching the encoding rule between two read/write operations on streams leads us to some difficult problems, one of which is how we should treat the current shift-state. What is the desirable behavior of wide character input/output functions after changing the value of the locale category, LC_CTYPE.

Discussions

(none)

Conclusion

Changing the value of the locale category LC_CTYPE between two read/write operations should cancel the validity of the data obtained by the latter operation. In other words, after changing value of the locale category LC_CTYPE that causes to switch the current encoding rule, the behavior of input/output streams, except ones that no read/write operations are applied, is undefined.

Thus, any locale value changing which causes to switch the encoding rule should occur before any read/write operations.

2.2 Wide Character Handling [3.1]

2.2.1 Type of the property argument of the set_wctype() function

Background

In the design process of character classification functions for wide characters and new character classes, only two functions like,

set_wctype(*property-name*)

and

is_wctype(*wc*, *property-id*)

which provide user- and/or dynamically- definable character classification are introduced, instead of defining all possible character classifications and their functions by collecting such character classifications all over the world.

Concerns

The question is raised why a property argument of the **set_wctype**() function is not one of integral types, but character pointer type.

Discussions

Another suggestion is that the *property-name* argument should be integral type just like the *category* argument of the **setlocale**() function.

Whereas, several intentions of the new functions are explained:

- A *property-name* or *class-name* would be extensible, covering user-definable character class.
- An integer type implementation of the *property-name* may lead to pre-definition or registration issues of the appropriate properties (classes) by implementors or certain authorities. It is not flexible but a constraint against freely or dynamically introducing a new character class as appropriate.
- A character pointer type enables a flexible binding of the property in the execution environment by a system command like the **localdef** command in POSIX.

Conclusion

The Working Group agreed that the *property-name* argument of the **set_wctype**() function be character pointer type, taking account of the flexibility.

Moreover, since several common character classes are expected among the all implementations, *the Working Group* decided to define and reserve the following eleven names as *the Standard* properties:

```
alnum   lower   xdigit
alpha   print
cntrl   punct
digit   space
graph   upper
```

2.2.2 Consideration of a single byte character classification extension

Concerns

There are new functions – the is_wctype() function and the set_wctype() function for "wchar_t" type which do not have "char" type counterparts.

Discussions and Conclusion

The Working Group discussed about possible "char" type counterparts, and came up with the idea – they are not necessary and the functionalities which such new functions may provide are covered by existing functions appeared in *the Standard*.

2.3 Input/Output [3.2]

2.3.1 Return values of wide character input/output functions

Concerns

Traditional specifications of (single byte) character input/output functions such as the getc() function and the putc() function are defined based on the fact (assumption) that a single byte character shall be covered by an integer. In other words, a sizeof(int) is always greater than a sizeof(char), or INT_MAX > CHAR_MAX. Some of these functions also require EOF which expands to a negative integral constant expression to indicate end-of-file. Therefore, a function which returns a (single byte) character is almost always defined as an integer type.

This is not true for wide character input/output functions, because a wide character (wchar_t) may be more than 2 bytes. An assumption that INT_MAX > WCHAR_MAX is no longer valid.

Note that in the Japanese Extension Specifications of SIGMA Project, the assumption INT_MAX >= WCHAR_MAX is made, assuming a wide character which covers a Japanese character as well as an English character is 2 bytes:

```
typedef unsigned short wchar_t;
```

and each function which returns a wide character is defined as an integer type. This approach has an advantage that the traditional conventions related to a integer, EOF handling are still valid. However, it is a limited solution.

To get a more general specification, there were two possible solutions:

i. Define as a long integer type.

ii. Define as a wchar_t type.

In *the Proposed Extension*, *ii.* was selected because a long int type is lengthy in a sense, it may prevent the implementation from tuning a size of wchar_t (since it is a typedef quantity) and memory required to support several kinds of wide character handling functions, and the assumption LONG_MAX > WCHAR_MAX is not always appropriate.

As a result, in this draft return values of wide character input/output functions are defined as a wchar_t type wherever the function returns a wide character.

2.3.2 WEOF

According to previous discussion, a **WEOF** macro is introduced to represent return value of wide character input/output functions at detecting the end-of-file. Although the **WEOF** corresponds to the **EOF** for (single byte) character input/output functions, it is no longer assumed to be a negative integer even if in many cases it will be just defined as:

```
(wchar_t)EOF
```

2.3.3 Null for wide character string literals

A wide character with all bits set to 0, is used to terminate a wide character string literal. There is no specific macro other than the **NULL** to represent it.

2.3.4 No new macro for a null pointer constant to the wide character

Concerns

In conjunction with a discussion about return values of wide character input/output functions and macros for wide character constants, there is a concern whether or not to define a new macro which corresponds to a null wide character constant or a null pointer constant of wide character type. For example, **WNULL** is suggested on the analogy of **WEOF−EOF** relation.

Discussions

According to *the Standard* [3.1.3.4, 3.1.4, 4.1.1, and 4.1.5],

- a wide character string terminates by a wide null character;
- a wide null character constant can be represented by **L'\0'**.

In addition to that, *the Standard* [3.2.2.3 and 4.1.5] implies that a **NULL** macro can be used as a null pointer constant of wide character type.

Therefore, there is no need to define a new macro for a null wide character constant or a null pointer constant of **wchar_t**.

Conclusion

The Working Group agreed *not* to add a new macro for **NULL** of **wchar_t** type.

2.3.5 New macro for an end-of-file constant returned by wide character functions

Concerns

As we propose wide character input/output functions such as the **getwc()** function, the **getwchar()** function, or the **fgetwc()** function, it is no doubt that the functions need to return end-of-file.

Can they return EOF as end-of-file?

The Working Group feels that *the Standard* assumes following facts on relations between character (or byte) input/output functions and **EOF**:

F1 A return value is treated as an **int**.

F2 On any actual implementation, a storage size of an **int** is truly greater than that of a **char**.

F3 **EOF** expands to a negative integer constant.

By analogy, on wide character input/output functions, their return value seems to be treated as an **int**, and their end-of-file macro shall be **EOF**.

But conflict occurs on some implementations on which integral type is represented in two's complement representation, a storage size of an int object is 16 bits, and **wchar_t** is defined as `unsigned short`. In this case:

maximum value of `wchar_t` > maximum value of `int`

can be true. Some wide characters will be treated as negative. Therefore, we cannot distinguish them from `EOF`.

The Working Group can solve this confusion if there exists the fact *F2*. To admit the *F2* may mean as follows.

F2-2 Every wide character, regardless of any locale specific, can be represented in positive `int`.

Since this request is too severe, we cannot solve the confusion with this approach.

Discussions

To solve this confusion, two suggestions were under examination by *the Working Group*.

- Suggestion 1:
  Instead of the *F1*, we introduce:

  F1-2 A return value is treated as a `long int`.

  However, this is not quite solution if

  maximum value of `wchar_t` > maximum value of `long int`

  is true.

- Suggestion 2:
  Instead of the *F1*, we introduce:

  F1-3 A return value is treated as `wchar_t`.
  However, another confusion occurs on some Japanese common implementations on which a storage size of an `int` object is 32 bits and `wchar_t` is defined as an `unsigned short`.

```
wchar_t      wc;

/*  ***  */

if( (wc=getwchar()) == EOF )
        /*
         * As the wc has unsigned short type,
         * it cannot be negative.
         */
```

Therefore, to admit the *F1-3*, we need the special wide character **`WEOF`**, which means end-of-file of wide character input/output functions stream and will expand to `(wchar_t)(-1)` or to `(wchar_t)EOF`.

Conclusion

The Working Group rejected the former solution because it has no expansion and is due to non-effective handling to wide character. As a result, *the Working Group* selects the latter solution, eventually a returned value from input/output functions for wide character is treated as a `wchar_t`, and there exists **`WEOF`** as end-of-file of wide character input/output functions.

2.3.6 Error handling during the code conversion in the functions

Concerns

There is an opinion that when an invalid data (violating of the encoding rule) is found during code conversion, wide character input/output functions should return the value indicating *invalid code error*.

Discussions

It may be useful to be informed the occurrence of an *invalid code error*. There is no reason to refuse this information.

There is another opinion that programmers may want to write a program where the erroneous data are skipped to recover from the error if the program detects it. But in order to implement error recovery it is necessary to impose some kind of restrictions upon the encoding rule. It is not suitable for *the Standard* to restrict encoding rule because there is a criteria that unnecessary restrictions should not be imposed upon the encoding rule in *the Standard*.

The Working Group should regard data which include erroneous code as totally unreliable. From this point of view *invalid code error* is similar to *read/write input/output error*.

Conclusion

Any wide character functions should detect an invalid code if it occurs. If they find it they should return any error value.

- For example, The `fgetwc()` function will return **WEOF**.
- It is implementation-defined whether there is an individual error value which ferror(and so on.) returns to identify *invalid code error*.

2.3.7 Insufficiency of the Standard multibyte functions in a shift-dependent encoding

Concerns

In *the Proposed Extension* we can find the following description:

> *The conversion (in wide character input/output functions) is done as if such functions call wctomb and/or wcstombs functions.*

But it is almost impossible to implement these wide character input/output functions with *the Standard* multibyte handling functions (mbtowc, mbstowcs and so on).

Discussions

If it may handle a shift-encoding rule, it is impossible for *the Standard* multibyte handling functions to treat two or more byte streams. See section 3.4 in this document.

On the contrary, each byte stream has its own FILE structure that keeps the current shift-state in code conversion.

Conclusion

In spite of the above description (in *the Proposed Extension*) we cannot implement code conversion mechanism in wide character input/output functions with only the multibyte handling functions in *the Standard*.

2.3.8 Redundant shift-sequences and a file position indicator for the stream

2.3.8.1 the fgetwc() function and file position indicator

Concerns

When the `fgetwc()` function is called for the multibyte character sequence which contains redundant shift-sequence, where should the file position indicator point to after its execution?

Discussions

Example 1:

```
SI SO SI SO SI XX YY SO
1  2  3  4  5  6  7  8  9
```

The file position indicator points to the top of redundant shift-sequence (position 1). After the execution of the fgetwc() function, to where the file position indicator advance.

Example 2:

```
SI XX SO SI SO SI SO SI YY SO
1  2  3  4  5  6  7  8  9  10 11
```

The behavior for such redundant shift-sequence is implementation-dependent. However, what we mean in the paragraph of the fgetwc() function in *the Proposed Extension*:

The file position indicator is advanced for one multibyte character obtained.

is to advance the file position indicator *one* wide character.

Conclusion

Example 1:

```
position 1 → fgetwc() → position 7 → fgetwc() →
position 8
```

Example 2:

```
position 1 → fgetwc() → position 3-9 (implementation
dependent) → fgetwc() → position 10
```

2.3.8.2 The `fgetws()` function and the file position indicator

Concern

To define the specification of the **fgetws()** function using that of the **fgets()** function as basis, the original description of the return values of the **fgets()** function in *the Standard*:

If end-of-file is encountered and no characters have been into the array, the contents of the array remain unchanged and null pointer is returned.

[4.9.7.2] the semantics of the term "no character" must be clarified.

That is, whether it is true – "no characters have been into the array" – when the current file position indicator points to the redundant shift-sequence like **SI SO SI SO SI SO** and advancing the position to seek the next character results to **EOF**.

Discussions

It is obvious for the implementations where there exists no wide characters which corresponds to the shift-sequences themselves.

Being mainly composed of "wide characters" which are stored in the array, the description (specification) should not state the points above, explicitly.

Conclusion

The description should handle "wide character" to be stored in the array, its "subject" as:

If end-of-file is encountered and no wide characters have been into the array, the contents of the array remain unchanged and null pointer is returned.

2.3.9 Names of the new conversion specifiers in the format character string

Concerns

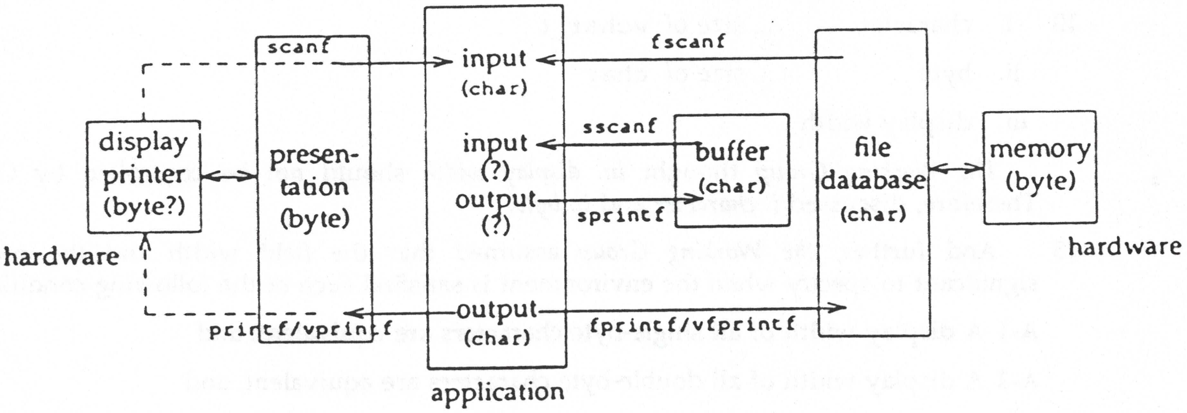
The conversion specifiers for wchar_t type were named as `%ws` and `%wc`, instead of `%ls` and `%lc`, neglecting an existing implementation of the similar specification. This is because the consistency with the name of type `wchar_t`.

2.3.10 Precision and field width interpretation of the wide character specifiers

Concerns

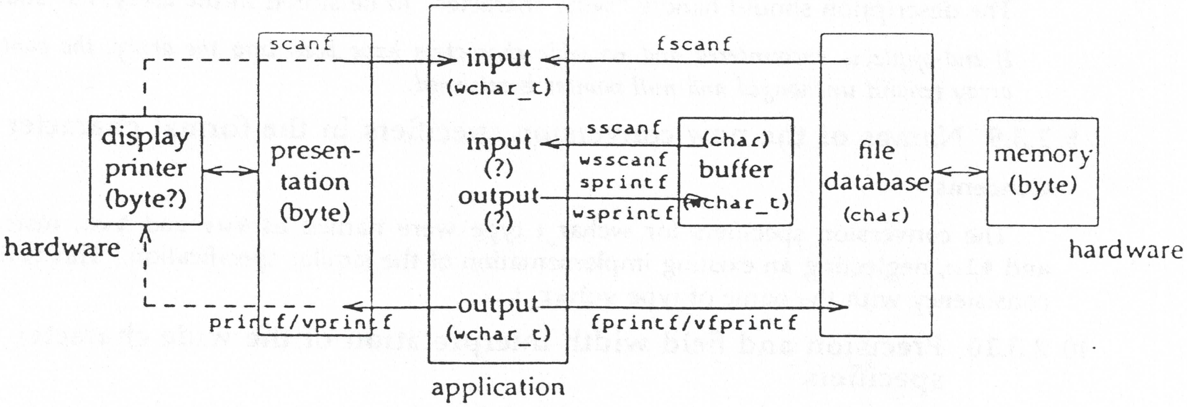
The Working Group has discussed the field width and the precision for `%wc`, `%ws` on formatted input/output functions, such as the `printf()` function and the `scanf()` function. The following shows the reason why *the Working Group* should have defined new semantics of field width and precision for `%wc`, `%ws` which are different from those for `%c`, `%s`.

As you can see in the following figure, the formatted input/output functions are the data processing between character-oriented environment which processes by character, and byte-oriented environment such as file, stream and peripheral which processes by byte. This means that a unit of field width and precision closely depends on the relationship between a unit of character and a unit of byte.



The Standard defines that field width and precision for `%c`, `%s` are counted by number of characters. In "single-byte character" world, a unit of character is equivalent to a unit of byte, since 1 character is usually represented by 1 byte.

On the other hand, input/output functions with `%wc` and `%ws` is the data processing between "character" and "byte". Note that "character" means multibyte character. Therefore, *the Working Group* has discussed whether a unit of field width and precision for `%wc` and `%ws` is "byte" or "character" or ...



Discussions

The functions which support %wc and %ws specifiers are:

- the printf() function, the sprintf() function, the fprintf() function, the vprintf() function, the vsprintf() function, the vfprintf() function
- the scanf() function, the sscanf() function, the fscanf() function
- the wsprintf() function
- the wsscanf() function

The following was the candidate as a unit of %wc and %ws.

i. character ... size of wchar_t

ii. byte ... size of char

iii. display width

The Working Group thought *iii. display width* should not be controlled by C language. Therefore, discussed *i. character* and *ii. byte.*

And further, *the Working Group* assumed that the field width and the precision are significant to specify when the environment is satisfied each of the following condition, *A* or *B*.

A-1 A display width of all single-byte characters are equivalent, and

A-2 A display width of all double-byte characters are equivalent, and

A-3 A double-byte character is twice of a display width of a single-byte character.

B-1 A display width of both all single-byte characters and all double-byte characters are equivalent.

For the environment which is not satisfied neither of above conditions, such as supports proportional font, we may specify the width based on a unit of display device and printer, such as dot or inch. This support will be included in future enhancement, when it is thought to be useful and significant.

i. The printf family functions
Generally, when user specifies the field width/precision in output functions, he will decide those according to the width which the data occupies on device. From a this point of view, it is significant for a unit of field width/precision to be byte, since these functions write to display, file and stream. Note that a unit of display width can be

considered to be equivalent to byte. However, regarding the `fprintf()` function and the `vfprintf()` function which write to display, the following should be considered. *The Working Group* has the display which takes the position for control character, such as shift-state switching code on state-dependent encoding, and also we have the display which does not take. If the display does not take the position for control character, it should be supported to specify the number of byte without control characters. Because user wants to specify the width, which the data actually occupies on device, as field width/precision.

Further, from the following application usage, field width and precision should be consistent with a width of display.

The field width and the precision are used to arrange any data in table of database, as follows:

```
1234567890123456789012345678901234567890
========================================
1  111111 abcXXYY     xxxxxx    xxxxxxxxx
2  222222 XXYYZZdefg  xxxxxxx   xxxxxxxxx
3  333333 abcdefg     xxxx      xxxxxxxxx

   1,2,..., a,b,c,...: single-byte character
   XX,YY,...,         : double-byte character
```

Since the precision is maximum number, converted value may be truncated when it is longer than precision. Then, the precision is useful on output as above. On the other hands, since the field width is minimum field width, all of converted value will be displayed, not truncated. So, user can specify the length, which is enough to put any data in one column, as field width. When the field width and the precision are specified by character, data width may be variable and unpredictable as you can see in the following.

Example 1: Output by "character" (7 multibyte characters)

```
123456789012345
abcD1D2D3D4
D1D2D3D4D5D6D7
abcdefg

a,b,c,...: single-byte character
D1,D2,...: double-byte character
```

However, if precision is specified by character, we can get the output that is consistent across the environment. That is, as you can see in the following, the character contents is same whether the environment supports shift-dependent encoding or not. From this point of view, the specification by characters seems to be useful. Of course, this is applicable only when output data can be truncated, that is, applicable for precision.

Example 2: Output by "character" (8 multibyte characters)

```
12345678901234567890
aD1bD2cD3dD4
a<D1>b<D2>c<D3>d<D4>

a,b,c,...: single-byte character
D1,D2,...: double-byte character
<,>       : control character for shift-state switching
```

ii. The `scanf` family functions
The width parameter on these input functions is useful for input of several data which are formatted in certain rule like a table, as shown below:

```
1234567890123456789012345678901234567890

1   111111 abcD1D2    xxxxxx    xxxxxxxxx
2   222222 D1D2D3defg xxxxxxx   xxxxxxxxx
3   333333 abcdefg    xxxx      xxxxxxxxx
```

So, at this kind of input processing, user will specify the width of input data in order to limit the length of character data which he wants to convert. Then, the specification by byte should be supported.

iii. The **wsprintf**() function and the **wsscanf**() function
These input/output functions include code conversion in between **wchar_t** and **wchar_t**. So, the specification by character, which is equivalent to by **wchar_t**, is proposed.

Conclusion

Except *i.* on above discussion, *the Working Group* has agreed. Regarding *i.*, that is, the **printf**() function, the **sprintf**() function, the **fprintf**() function, the **vprintf**() function, the **vsprintf**() function, and the **vfprintf**() function, there was the following proposal.

	Proposal-1	Proposal-2	Proposal-3
printf: field length	byte	byte	byte/char
precision	byte	byte/char	byte/char

byte: specified by byte

char: specified by character

byte/char: specified by byte or character, switched by # flag

The result of vote was:

Proposal-1:	1
Proposal-2:	7
Proposal-3:	5

According to above and more discussion, the conclusion of *the Working Group* was proposal-2. Therefore, proposed the following semantics of the field width and the precision for **%wc**, **%ws**.

functions	*field width*	*precision*
`printf()`, `sprintf()`, `fprintf()`, `vprintf()`, `vsprintf()`, `vfprintf()`	byte "`char`"	byte "`char`" (as default) character "`wchar_t`" (with #flag)
`scanf()`, `sscanf()`, `fscanf()`	byte "`char`"	
`wsprintf()`	character "`wchar_t`"	character "`wchar_t`" (even if #flag)
`wsscanf()`	character "`wchar_t`"	

Further Discussions

Even though *the Working Group* has once concluded this proposal, there may be required some discussions in order to make the functionality be more useful. So, *the Working Group* will take the more consideration and reflect to next revision of the draft if change can be made.

2.3.11 Scanset discussion for multibyte characters

Concerns

In the review process of completeness of *the Proposed Extension*, it is pointed out that multibyte or wide character discussion is missing regarding "scanset" of the `scanf()` function family.

In order to complete a new paradigm using wide character functions, it is suggested to discuss this topic.

Discussions

The Standard says [4.2.6.2]:

> *The format shall be a multibyte character sequence, beginning and ending in its initial shift-state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither* `%` *nor a white-space character); or a conversion specification.*

On one hand, in the specifiers description of *the Standard*:

`%s` *Matches a sequence of non-white-space characters. (*118)*

`%[` *Matches a nonempty sequence of characters (*118) from a set of expected characters (the scanset).*

`%c` *Matches a sequence of characters (*118) of the number specified by the field width. ...*

...

_____ **118 No special provisions are made for multibyte characters.*

The above shows:

- The scanset representation cannot handle a multibyte character as a single "character".
- Therefore, a complement (circumflex `^`) expression of the scanset cannot correctly specify a "character" group which may include multibyte characters.

Consider a example: `%[^0123456789]`. Does this match multibyte characters which may appear in the execution environment?

These issues imply incompleteness of multibyte character features in *the Standard*, and thus it is suggested in *the Working Group* that *the Proposed Extension* should cover the issues.

On the other hand, due to lack of discussion time, several members proposed to postpone re-examining the issues until the next draft.

Meanwhile, it is noticed that these issues are also found in the strscn() function and the strcscn() function. For example, there is no discussion in *the Working Group* about a complement character set in the strcscn() function from a multibyte point of view.

Although the wcscn() function and the wcscscn() function specifications in *the Proposed Extension* give one of the solutions to the problems, the discussion is not completed.

Conclusion

The Working Group decided to postpone the discussions until the next revision of the draft.

2.3.12 Field width of shift-sequences for the %wc specifier of the fscanf function

Concerns

Should define the default field width for %wc on formatted input library functions. As you know, the byte length for one multibyte character is variable. For state-dependent encoding, some shift-sequences are encountered in the code sequence in order to switch the shift-state.

Discussions and Conclusion

The Working Group decided to postpone the discussions until the next revision of the draft.

2.3.13 Field width for the %ws specifiers and its portability

The Working Group discussed whether the field width for %ws should be interpreted as number of byte or number of character.

The word "characters" here means not only "single-byte characters" but also means "multibyte characters".

Interpretation of both field width and precision as number of character

- Basis of interpretation
  If a programmer specifies field width and/or precision in consideration of printing format, it has no meaning to interpret these as number of byte. Even if each output devices have their own character sizes, it seems that it is intention of a programmer that the field width and/or precision should be interpreted as number of character.

- Problem
  It is necessary for a programmer to be conscious of correspondence between the wide characters and their sizes on specific output devices.

Interpretation of both field width and precision as number of byte

- Basis of interpretation
  When field width and precision are interpreted as number of character, character sizes are different according to the output device. So it seems that field width and/or precision are available in such cases when a programmer would like to be conscious of buffer size for input/output.

- Problem
  When a programmer would like to be conscious of output format, the programmer should calculate the character sizes by using number of byte.

Conclusion

The above two discussions have both merits and demerits, so we agreed with the specification proposed for more details, refer to 2.3.10.

2.3.14 Overhead of the %ws support

There was discussion that overhead for library is increased with supporting `%ws` conversion specifier. But programming language C is utilized internationally and utilization of multibyte is indispensable on the occasion of internationalization. Now, for multibyte code only 5 functions are supported in *the Standard* but these functions are not enough for handle multibyte code systematically. Therefore, we proposed standardization of additional functions from point of view that handling of multibyte code is important.

2.3.15 %s vs. %ws

Though these are the same in function, we consider the value of `%ws` is enough for existence to handle multibyte systematically as `wchar_t` type. So we proposed `%ws` conversion specifier.

2.3.16 sprintf/scanf vs. wsprintf/wsscanf

Concerns

The following two concerns were raised during a wide character extension discussion about **sprintf/sscanf** family.

- Format extensions like **%wc** and **%ws** are sufficient?
- What about an extension of the target memory to a wide character memory array?

Discussions

- *The Proposed Extension* aims at establishment of a new programming paradigm based upon the proposed wide-character-oriented functions, in order to ensure international portability of the C programs that use this extension.
- The **sprintf()** function and the **sscanf()** function are frequently used and important as well as the **printf()** function and the **scanf()** function. Especially, programmers prefer the **sscanf()** function to the **scanf()** function. Therefore, wide character variants of the **sprintf()** function and the **sscanf()** function are requested by several C communities.
- Wide character functions corresponding to the **sprintf()** function the **sscanf()** function have been proposed and defined by both *the "Japanese UNIX System Advisory Committee"* (April, 1985) and the *SIGMA project* (SIGMA OS Japanese Extension Specification, March 1990).
- There is no strong objection against the **wsprintf()** function and the **wsscanf()** function.

Conclusion

The Working Group decided to add the **wsprintf()** function and the **wsscanf()** function to *the Proposed Extension*.

2.4 Future Library Directions [3.6]

2.4.1 Naming conventions of the wide character functions

Concerns

If we introduce library functions for multibyte character and/or multibyte string processing without any naming rules, it will make both users and implementors of such library functions confused. *The Working Group* may have to establish some naming convention for such functions and to follow it.

Discussions

i. *The Standard* describes on the length of identifiers as: "*... the implementation may further restrict the significance of an external name (an identifier that has external linkage) to six characters ...*" [3.1.2].

The function names of the multibyte library which we are going to define should be identified with first 6 letters.

ii. *The Standard* describes that "function names that begin with `str`, `mem`, or `wcs` and a lower-case letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the `<string.h>` header." [4.13.8]. The functions names of the multibyte library which we are going to define should be prefixed with `wcs`, as much as it is possible.

Conclusion

The Working Group introduces following conventions for the function names of the multibyte library:

`<ctype.h>` the wide character functions which has single byte counterpart is named, adding `w`, to mean wide character, following to the `is` or `to`.

Examples		
`isalnum`	☞	`iswalnum`
`tolower`	☞	`towlower`

The `set_wctype()` function and the `is_wctype()` function do not follow this rule, since these are new and have no single byte counterparts.

`<stdio.h>` Considering the rule *ii.* above, the names of the get/put family functions for wide character have `wc` and/or `ws` following to the get/put. If we follow the rule 2. above, the wide character counterpart for the `fputs()` function would be the `fputwcs()` function, and it could not be distinguish with the `fputwc()` function for their first 6 letters. With the similar reasons, wide character counterparts for the `sprintf()` function and the `sscanf()` function are preceded by a `w`.

Examples		
`fgetc`	☞	`fgetwc`
`fputs`	☞	`fputws`
`sprintf`	☞	`wsprintf`

The the `printf()` function, the `fprintf()` function, the `sprintf()` function, the `scanf()` function, the `fscanf()` function, the `sscanf()` function, the `vprintf()` function, the `vfprintf()` function, and the `vsprintf()` function have just been added new conversion specifiers – `%wc` and `%ws`, and there existing functionalities are not changed.

`<string.h>` the function which has traditional counterpart is named by replacing corresponding `str` with `wcs`.

Examples		
`strcpy`	σ	`wcscpy`
`strcat`	σ	`wcscat`

`<time.h>` Same as `<string.h>`.

2.4.2 Header files

Concerns

The location of the function prototypes for the multibyte library functions should be specified by the extension. Existing implementations have different definitions.

Discussions

Following ideas were discussed:

i. Following the naming convention, `<wctype.h>`, `<wstdio.h>`, `<wstring.h>`, and so forth should be introduced.

ii. All the function prototype should be located in the existing header file `<stdlib.h>` where the `wchar_t` type is defined.

iii. Introduce a new header file, and put all the prototypes for the library in there.

iv. Each prototype for the library functions should be located in the existing header files which correspond to the single byte counterparts.

If we choose *i.* above, the relationship between new header files with traditional ones, especially their dependencies, becomes complicated. For example, user may have to include two header files in prior to `<wstdio.h>` in some implementations, as below:

```
#include <stdlib.h>
#include <stdio.h>
#include <wstdio.h>
```

If we do *ii.* above, the user of the header file `<stdlib.h>` who doesn't need `wchar_t` but the others in the header file must incorporate the large set of prototypes of the multibyte library functions. *iii.* above appears economic, but not so intuitive.

In the past, there appears opposition to add new things to the standard header file, especially to `<stdio.h>`. In *the Standard*, contents of the standard header files appears to be dealt tolerantly in the various environments; some data types like `wchar_t` are declared in both `<stddef.h>` and `<stdlib.h>`, for example.

Conclusion

The Working Group will not define new header files for the multibyte library functions, but to add declarations and definitions of prototypes to the existing header files.

The Working Group does not assume that `<stdlib.h>` and/or `<stddef.h>` are included before the corresponding header file, even though the multibyte functions defined in it refer to `wchar_t` type. On the other hands, it should be an implementation issue to avoid the multiple definitions of data types in various header files, when these are included to a file, just like data types in `<stdio.h>` and `<stdlib.h>`.

3. Rationale - not corresponding to SC22/WG14/N104

3.1 Default locale at program startup

Concerns

The Standard describes that an application program should behave as if `setlocale(LC_ALL, "C");` is executed at start up time [4.4.1.1]. However, it must be useful for real programs to expect `setlocale(LC_ALL, "");` is executed instead of above. That is, the locale at start up time of application programs is the default locale of the system.

Discussions

The Working Group consulted the meeting minutes of *X3J11* on this issue to avoid duplication of discussions on the resolved matter [X3J11/85-092, X3J11/86-109, X3J11/86-125, X3J11-86-145, X3J11/86-151]. As result, *the Working Group* found that the current specification was decided by a ballot, while no real discussions prior to the ballot are recorded in the minutes.

Possible concerns must be on the behavior when the default locale cannot be obtained.

Conclusion

Since it appears that the issues was discussed in *X3J11* great deal, *the Working Group* decided to just follow the current specification. However two questions on the specification below were raised:

i. There is a description in *the Standard*:

> *... at program startup, the equivalent of setlocale(LC_ALL, "C"); is executed.*

It is not clear that the phrase – *"is executed"* means something like – *"shall be executed"*, or not.

ii. The definition of the "C" locale – *"minimal environment for C translation"* is not clear.

3.2 No wide string counterparts for the file handling functions

Concerns

During the survey of multibyte extension to *the Standard* functions which take character and/or string parameters as their arguments, it is asked whether or not the functions which have "filename" argument like the **`remove()`** function [4.9.4.1], the **`rename()`** function [4.9.4.2], the **`tmpname()`** function [4.9.4.4], the **`fopen()`** function [4.9.5.3], and the **`freopen()`** function [4.9.5.4] should be extended as corresponding wide character functions.

Discussions

- Positive discussions:

P1 "A Proposal to the ANSI C" produced by *the Japan C Language Committee* in 1987 addresses several extended functions as follows:

```
— l_remove(const letter *filename);
— l_rename(const letter *old, const letter *new);
— l_tmpname(letter s*);
— l_fopen(const letter *filename, const char *mode);
— l_freopen(const letter *filename, const char *mode,
        FILE *stream);
```

[X3J11/87-064]

Note: `letter` corresponds to `wchar_t` in *the Standard*.

P2 Aiming at a full set of `wchar_t` functions corresponding to traditional `char` functions, these functions like the **`wfopen()`** function, the **`wfclose()`** function, and so on would be supported even though their priorities are not high.

P2′ Since *the Working Group* endorses a programming paradigm that all "character" strings and arguments should be handled as wide character arrays and manipulated using wide character functions, such philosophy should be applied to these filename arguments and functions.

- Negative discussions:

N1 Because a file name is an implementation-dependent system (kernel) interface issue and the name space issues are out of *the Standard* scope, there is no need to introduce new `wchar_t` functions that handle filename as a wide character string.

N2 A genuine intention of the enhancement of `wchar_t` functions in *the Proposed Extension* lies in providing enriched functions that directly contribute to ease of multibyte character processing in programs, as well as single byte character processing.

Therefore, there are no good grounds for adding `wchar_t` filename functions.

Conclusion

The Working Group agreed upon *not* to add new functions which take filename as in `wchar_t`.

3.3 No extension to harden the Standards multibyte functions in a shift-dependent encoding

Issues

The detailed studies in *the Working Group* shows that the specification of the Standard multibyte functions – the **`mblen()`** function [4.10.7.1], the **`mbtowc()`** function [4.10.7.2], the **`wctomb()`** function [4.10.7.3], the **`mbstowcs()`** function [4.10.8.1], the **`wcstombs()`** function [4.10.8.2] are not sufficient in a state-dependent encoding.

It is clearly pointed out that in order to ensure a correct behavior of the functions even in any state-dependent encoding, necessary are several specification changes like introducing new argument which can hold, notice and specify the current state information of the encoding.

Another issue is whether *the Working Group* should go further by enhancing the **`mbxxx()`** functions family interfaces, as well as proposing enriched wide character functions.

Discussions

To resolve the problems, there are several options:

i. Change *the Standard* multibyte function interfaces;

ii. Not change the interfaces, but add a certain restrictive description for state-dependent encoding cases;

iii. Neither change nor add to the current specification. Rather, endorse wide character functions in *the Proposed Extension* instead of the multibyte functions in *the Standard*

The option *i.* is evaluated as the most difficult solution, because it requires a lot of discussions about the state-dependent encodings and at the same time there may be large side effects to the existing function interfaces and their programming style.

The options *ii.* and *iii.* are considered as feasible solutions.

Decision

The Working Group believes that it is inadvisable to harden multibyte handling functions of *the Standard* in the state-dependent encoding environment.

Therefore, *the Working Group* decided *not* to enhance the interfaces, rather to take the options *ii.* and *iii.* above.

3.4 Insufficiency aspects on code conversion in the Standard

Concerns

i. The mbstowcs() function and the wcstombs() function can handle more restricted class of shift-encoding rules than the mbtowc() function or the wctomb() function.

ii. The mbtowc() function and the wctomb() function cannot handle two or more data streams simultaneously.

Discussion

- We should regard the mbstowcs() function and the wcstombs() function as separated from the mbtowc() function and the wctomb() function. This is because the former can be handle more restricted class of shift-encoding than the latter.

  Only the strings that start on initial shift-state can be fed to the mbstowcs() function. This means that you should find where initial shift-state is with no help of any library functions, only which should know about the encoding rule.

  Because the result multibyte string which the wcstombs() function returns is always SITI (Start Initial Terminate Initial, it means the property of multibyte character string that starts and terminates on initial shift-state), the concatenated data may include redundant shift sequences.

  If the encoding rule accepts no such redundant shift-sequences, you should write codes to throw the redundant sequences away so that the resulting multibyte strings be interpreted according to the encoding rule. Generally speaking, this is not feasible to implement, because it is very tremendous to interpret on a shift-encoding in general.

  On the contrary, if the encoding rule allow us to easily determine where initial shift-state is (for example, encoding on which initial state occurs at each end-of-line.), the mbstowcs() function and the wcstombs() function may be useful.

- Some additional regards should be needed if you would like to handle two or more text data streams (NOTE: the meaning of the phrase "text data streams" above is different from the term "text stream" in *the Standard*) with the mbtowc() function and the wctomb() function on some shift-encoding. In such a case you will find it is necessary to hold one current shift state for each of these streams. As for the mbtowc() function and the wctomb() function, because each of them are expected to hold one shift-state in its static storage hidden from the programmers, they can handle single text data stream at a time.

  They are not sufficient for applying two or more text data streams. In order to solve the problem, we should introduce a new argument pointing to the current state of code-conversion into arguments of the mbtowc() function and the wctomb() function.

- We should note that there is a policy of *the Working Group* not to touch the description in *the Standard* itself.

Conclusion

We will point out the problem in multibyte handling functions of *the Standard* in this Rationale. We will not propose a plan to modify the specification about multibyte handling functions in *the Standard*.

3.5 A code value of the wide character data

Concerns

The Standard specifies the following [4.1.5] regarding the `wchar_t` code value for basic character set:

... each member of the basic character set defined in 2.2.1 shall have a code value equal to its value when used as the lone character in an integer character constants.

As the meaning of above *the Standard* statement, the following two meanings may be expected.

i	`'A' == L'A'`	Supposing `'A'` has code value `x'AA'`, `L'A'` is `x'00..0AA'`.
ii	`(char)'A' == (char)L'A'`	Supposing `'A'` has code value `x'AA'`, `L'A'` is `x'**..*AA'`. Note that `'**..*'` can be any code value.

Discussions

(none)

Conclusion

The Working Group recognized C compiler is implemented based on interpretation *i*. above. That is, `wchar_t` value of the basic character set pre-pends code value zero to its value.

3.6 String concatenation problem about shift-sequence handling

Concerns

In a *shift-encoding environment*, what is the result of the `strcmp()` function.

```
char *s1 = "XXYY";        /* SI XX YY SO */
char *s2 = "XX""YY";      /* SI XX SO SI YY SO */
int  result;

result = strcmp(s1, s2);
```

Discussion

During the translation phase, the character code in the physical source files are mapped to the source character set and the members of the source character set in string literals are translated in to the execution character set. String concatenation for the pointer `s2` above, then is executed [2.1.1.2].

On the other hand, shift-sequence (`SI` or `SO`, here) can be treated as element of the multibyte character set [2.2.1.2]. Therefore, shift-sequences in the string literal remains after the string concatenation as an individual elements of character set.

Conclusion

In general, the result of the `strcmp()` function as shown above will be false, for shift-encoding environments.

3.7 % character in the format string of the fprintf() function

Concerns

In the 7 bit encoding rule according to ISO2022 (Code Extension) where there are two character set, ASCII and JISX0208-1983 (Japanese characters), there are two "percent" characters, one is in ASCII, the other is in JIS characters.

If there are two or more "percent" characters under some encoding rule, how should they be treated in format argument of the `fprintf()` function?

Discussions

The Standard clearly tells about the question. It tells that we should treat only one in the basic execution character set (in the above example, one in ASCII) as escape character of format argument.

Conclusion

The Standard expect that:

the special characters in format argument is ones in the basic execution character set.

3.8 Effects of the setlocale() function

Concerns

In *the Standard*, the functions whose behavior is affected by the value of current locale are as follows:

- the functions which are influenced by LC_COLLATE
  the `strcoll()` function, the `strxfrm()` function
- the functions which are influenced by LC_CTYPE
  the character handling functions except the `isdigit()` function and the `isxdigit()` function.
- the functions which are influenced by LC_TIME
  the `strftime()` function

In *the Proposed Extension*, the following functions are influenced by the locale additionally.

- the functions which are influenced by LC_COLLATE
  the `wcscoll()` function, the `wcsxfrm()` function
- the functions which are influenced by LC_CTYPE
  the `iswxxx()` functions family, the `towxxx()` functions family
- the functions which are influenced by LC_TIME
  the `wcsftime()` function

Discussions

The Working Group agreed it should be written apparently in *the Standard* that the locale specified by the `setlocale()` function influenced in *the Proposed Extension*.

Conclusion

The following contents should be added to the next revision of *the Standard*.

- Adding the `wcscoll()` function and the `wcsxfrm()` function to the functions influenced by LC_COLLATE

- Adding the **iswxxx**() functions family and the **towxxx**() functions family to the functions influenced by LC_CTYPE (contents of footnote 98)
- Adding the **wcsftime**() function to the functions influenced by LC_TIME.

A. APPENDIX: Conversation with X/Open Internationalization Working Group

A.1 Comments/Requests from X/Open

This memo was distributed at ITSCJ/SC22/C WG/SWG meeting on 20th December 1989 in Japanese, and is now translated into English to help reading replies from the group.

From: X/Open Internationalization Working Group
To: ITSCJ/SC22/C WG/Sub-working Group on Multibyte Support Functions
Subject: Comments and Proposals on the Draft Proposed Multibyte Character Support Extension for ANSI C (Draft 1.2)
Date: Wed, 20 Dec 89 12:23:45 +0900

A.1.1 Comments

A.1.1.1 On 2.3.1 Streams

The reason why the description of the `fgetpos()` and `fputpos()` functions were separate from those of the `ftell()` and `fseek()` is unclear. You had better to rewrite the paragraph.

A.1.1.2 On state-information of the state-dependent encodings

The `mbtowc()` function cannot be applied to multiple multibyte character sequences at a time; state-information should be managed at least *per stream*. Please forward this comment to ISO.

A.1.1.3 On multibyte characters and single-shift characters

(also from X/Open Kernel/RealTime Working Group)

It is not clear that how implementors of the ANSI C can read the description of multibyte characters when they want to support encodings which use single shift characters. Please forward this comments also to ISO and/or ANSI.

A.1.1.4 Error conditions for input/output operations

In your proposal, input/output functions return WEOF, when they encounter errors, during the input/output operation, while it is hard for users to distinguish real input/output errors with the errors during the conversions like the `mbtowc()` function does.

Even though you don't plan to define error conditions (`errno`) in your proposal, you'd better to describe the fact – there exist two types of errors during input/output operation.

A.1.2 Proposals

A.1.2.1 Change Requests

Please consider to add following changes in your proposal to ISO.

- **The "," flag character for `printf()`**
  Please refer the attachment [XoTGinter:547].
- **`strftime()`**
  Please refer the attachment [XoTGinter:546].
- **`wcsftime()`**
  Same with the change requests on the `strftime()`.

- Printing Position vs. Bytes in `printf()/scanf()` family
  Please refer the attachment [e-mail from Tom Yap].

A.1.2.2 Addition Requests

Please consider to add following new functions in your proposal, to synchronize with X/Open function set.

- **`strfmon()`**
  Please refer the attachment [XoTGinter:548].

- **`strptime()`**
  Please refer the attachment [INT/1289/12].

- **`mbfresync()`** †
  Skip the data to the next "valid spot" on the stream, after a read error occur during the read of multibyte character, and returns the number of bytes skipped. Implementations of the function may skip all the bytes from the input stream till they encounter end-of-file.

- **`mbresync()`** †
  The buffer version of the **`mbfresync()`** function, below.

- **`mbwidth()`** †
  Please refer the attachment [**`mbcol()`** in XoTGinter:594, **`mbscol()`** in XoTGinter:597]

- **`nl_cat()`** †
  which corresponds with:

```
        char *p;
        ...
        (*p);
```

- **`nl_cadvance()`** †
  which corresponds with:

```
        char *p;
        ...
        (*p++);
```

- **`nl_mbat()`** †
  which corresponds with:

```
        char *mbcp;
        ...
        (*mbcp);
```

  where `mbcp` is the pointer to a multibyte character.

- **`nl_mbadvance()`** †
  which corresponds with:

```
        char *mbcp;
        ...
        (*mbcp++);
```

  where `mbcp` is the pointer to a multibyte character, and is advanced to the next multibyte character [**`mbsadvance()`** in XoTGinter:601].

† The name of the function might be changed later.

- **nl_mblen()** †
  The stateful mblen() function, taking the shift-state as another argument [s_mblen() in XoTGinter:593].

- **nl_mbstowcs()** †
  The stateful **mbstowcs()** function, taking the shift-state as another argument [s_mbstowcs() in XoTGinter:591].

- **nl_mbtowc()** †
  The stateful **mbtowc()** function, taking the shift-state as another argument [**s_mbtowc()** in XoTGinter:592].

- **nl_wc()** †
  which corresponds with:

  ```
  wchar_t *wcsp;
  ...
  (*wcsp);
  ```

- **nl_wcsadvance()** †
  which corresponds with:

  ```
  wchar_t *wcsp;
  ...
  (*wcsp++);
  ```

- **nl_wcstombs()** †
  The stateful **wcstombs()** function, taking the shift-state as another argument [s_wcstombs() in XoTGinter:598].

- **nl_wctomb()** †
  The stateful **wctomb()** function, taking the shift-state as another argument [**s_wctomb()** in XoTGinter:595].

- **wcsfmon()** †
  The wchar_t counterpart of the **strfmon()** function.

- **wcsfresync()** †
  The wchar_t version of the **mbfresync()** function.

- **wcsresync()** †
  The wchar_t version of the **mbresync()** function.

- **wcstol()**
  The wchar_t counterpart of the **strtol()** function.

- **wcstoul()**
  The wchar_t counterpart of the **strtoul()** function.

- **wcwidth()** †
  Please refer the attachment [**wccol()** in XoTGinter:600, **wcscol()** in XoTGinter:599]

A.1.3 Misc.

- `mbftowc()`
  In X/Open, the `mbftowc()` function was withdrawn.

A.2 Reply to X/Open

From: IPSJ/ITSCJ/SC22/C WG/SWG (Japan)
To: X/Open Internationalization Working Group
Subject: Response for "Comment and Proposal" of X/Open I18N WG
Date: Fri Mar 9 20:52:42 JST 1990

Thank you for giving us your comments and change requests to our "Draft Proposal Multibyte Support Extension of ANSI C" (Draft 1.2). We've modified our draft, and submitted the Draft 2.0 to the ISO/SC22/WG14 as for addendum of the ISO 9899 (which is now DIS 9899). It will be discussed at the next WG14 meeting in London, on 18th and 19th June 1990.

We plan to summarize all the comments and proposals, we received officially, including what we did not incorporate into the Draft, and to attach them as appendices bringing into the London Meeting.

[Summary]

We have admitted your four comments and one addition request (`wcstol()` and `wcstou()` in 2.2), and modified the Draft and its Rationale. However, we did not include rest of your proposals to our Draft. We categorize the reasons why we did so into three types below:

i. The fundamental purpose of our work is to promote international portability of the programs written in ISO C as an computer language. We feel that the function might be powerful for internationalization and so important. It does not, however, directly concern multibyte/wide character processing (e.g., monetary, time & date, character width (column)).

   We have focused on extending features for multibyte/wide character handling in our Draft, and indeed have not enough time to discuss to establish a consistent view for features other than multibyte/wide character handling. Due to the lack of such view, at this point we cannot commit whether we will incorporate the function in our Draft in the future, to ISO.

ii. We don't think that the function fits for the basic concept of ISO C, and don't incorporate it into our Draft (e.g, "resync" and code error recovery).

   In some encoding rules, "resync" functions might be meaningful and usable. In other encoding rules, such as shift encoding, we cannot imagine the behavior of the "resync" functions clearly.

   To introduce the "resync" functions into ISO C, it would be necessary to impose a certain constraint upon the specification about encoding rules in order to refuse encoding rules that do not allow "resync" functions. This constraint does not fit the slogan – encode independence – in ISO C.

iii. The function belongs to the other programming style than we assume in our Draft (e.g., `nl_cat()`, `nl_advance()`).

   The programming style which the functions in our Draft provide:

- All the multibyte characters are converted to corresponding wide characters by the functions, so that programmer do have no needs to directly handle multibyte characters in his/her programs;
- All the character handling are done in wide character representation.

On the other hand, functions like "at" and "advance" over the multibyte characters introduce different style of programming, and may enforce programmers to concern about length of each character, error recovery in the application side, and whatever complicated depending upon encoding rules. We believe all such functionalities which "at" and "advance" may have are also provided by the functions in our current Draft over the wide characters, in much more portable and elegant way.

[Resolutions]

1.1 (On 2.3.1 Streams)

We adopted; we modified the corresponding part of our Draft (Draft 2.0).

1.2 (On state-information)

We adopted; we incorporated the point into our Rationale.

1.3 (On single-shift characters)

We adopted; we will modify the Draft (>2.0) and/or its Rationale.

1.4 (Error conditions)

We adopted; we have a sentence in the Draft 2.0 about it.

The "code conversion error" should be taken into account in any implementations as you mentioned. We have the statements in our Rationale as:

- The "code conversion error" itself should be detected;
- It is implement-defined whether "code conversion error" and "(physical) I/O error" can be treated as different kind of ones.

2.1 (Change Requests)

- The `printf()` flags corresponding to monetary feature
  Rejected – type 1.
- `strftime()`, `wcsftime()` – about field width and precision width
  Rejected – type 1.
- field/precision width should be by column in `printf()`, `scanf()`
  Rejected – type 1.

2.2 (Addition Requests)

- `strfmon()`
  Rejected – type 1.
- `strptime()`
  Rejected – type 1.

- **mbfresync()** – "resync" on multibyte string
  Rejected – type 2.

- **mbresync()**
  Rejected – type 2.

- **mbwidth()** – column width
  Rejected – type 1.

- nl_cat(), nl_cadvance(), nl_mbat(), mbsadvance()
  Rejected – type 3.

- **s_mblen(), s_mbstowcs(), s_mbtowc(), s_wctomb(), s_wcstombs()**
  Rejected – type 3.

- **nl_wc(), nl_wcadvance()**
  Rejected – type 3.

- **wcsfmon()** – wide character version of strfmon()
  Rejected – type 1.

- **wcsfresync(), wcsresync()** – "resync" for **wctomb**-type conversion functions
  Rejected – type 2.

- **wcstol(), wcstoul()**
  Incorporated.

3. Misc.

We added the **vwsprintf()** function to our draft 2.0.

We do not have time to review your recent change requests sent by Hirasawa-san; we will review them in our future meetings.

Information Technology Standards Commission of Japan

ITSCJ

Information Processing Society of Japan
Kikai Shinko Building No 3-5-8 Shiba-Koen Minato-ku Tokyo 105 JAPAN

TEL +81 3 431 2808
FAX +81 3 431 6493
TX 02425340 IPSJ J

WG14/N115

Multibyte Support Extension

To Make the Language C International

Norihiro Kumagai
1990-6-18,19

1 Our Proposal

* We propose
the Multibyte Support Extension(MSE)
to be a New Work Item of the WG14.

* Documents:

1. Draft Proposed Multibyte Support Extension of ANSI-C,

2. Rationale of Multibyte Support Extension of DIS9899,

by SC22/C WG IPSJ/ITSCJ Japa

WG14/N115

X3J11/90-061

2 Character Processing for Asian Language

Many Many Characters

e.g Over 6000 Chars (in Japan)

⇓

We should handle large char sets for world-wide application.

⇓

Byte (≈ 256) ≠ Character (more than 6000)

⇓

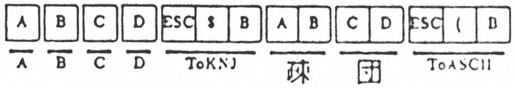
⟶ Multibyte Characters

3 Concerns on Multibyte Character Processing

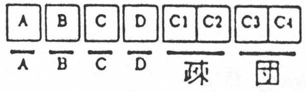
- Shift encoding vs. non shift encoding.
- Interpretation of byte sequence.
  - Meaning of a certain byte depends on its position.
- Difficult/impractical to concatinate/splitting strings.

4 Shift Encoding vs. Non-Shift Encoding

- Shift Encoding – multiple states
  ('A' may have different meaning)

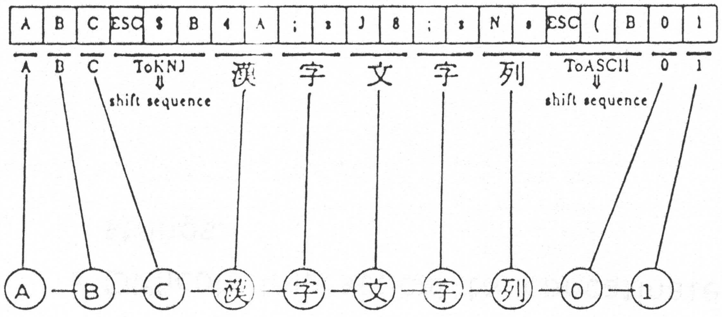


- Non-Shift Encoding — mono state
  (not so easy to interpret)
  (but easier than in shift encoding)



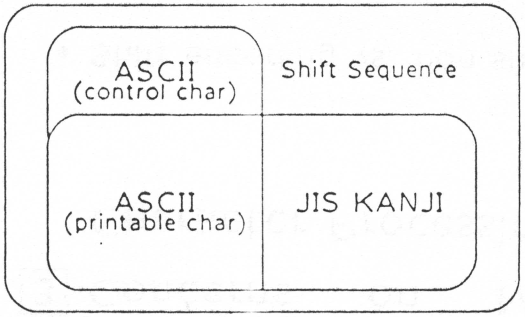
5 Byte Encoding & Character Set

Byte Sequence:

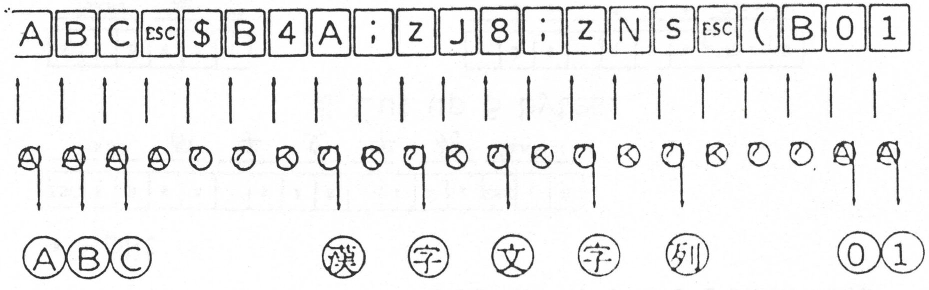


Character Sequence:

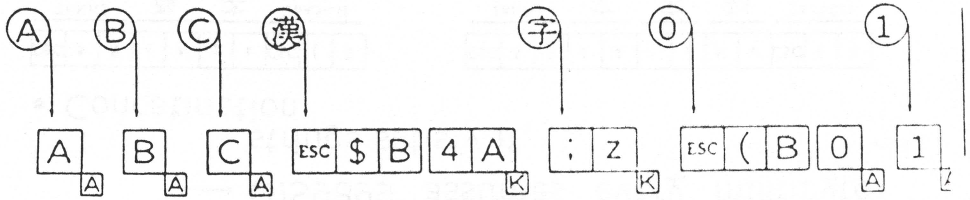
- An Example of Execute Char Set



6 Interpretation of Byte Sequence



Convert to Byte Sequence

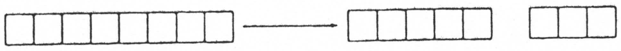


⟶ need to hold CSS (Current Shift State)

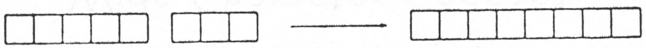
7 String Manipulation

Two Primitive Operations

- Split



- Concatination



⟶ These primitives are significant under the rule of 1 Char = 1 Byte

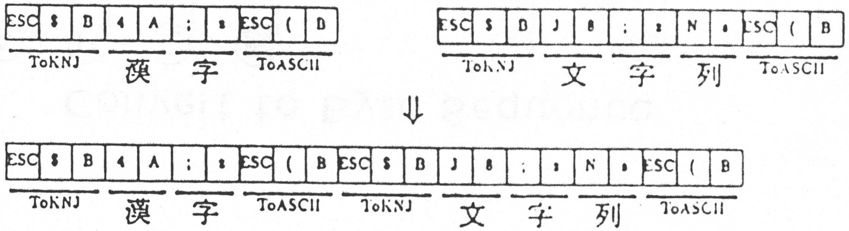
Under Multibyte Character and/or Shift Encoding, There are Several Problems

8 String Manipulation in Shift Encoding

- SITI Start with Initial Shift State, and Terminates with Initial Shift State, too.

  ⟶ DIS9899 assumes every multibyte strings satisfy it.

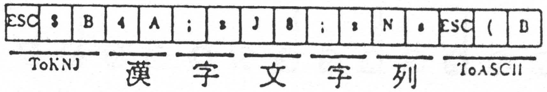
- Concatination



ESC $ B 4 A ; : ESC (D
ToKNJ 漢 字 ToASCII
ESC $ D J 8 ; : N , ESC (B
ToKNJ 文 字 列 ToASCII
⇓
ESC $ B 4 A ; : ESC (B ESC $ B J 8 ; : N , ESC (B
ToKNJ 漢 字 ToASCII ToKNJ 文 字 列 ToASCII

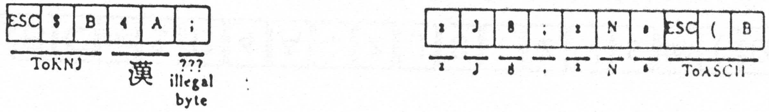
⟶ Redundant Shift Sequences.

- Split



ESC $ B 4 A ; : J 8 ; : N , ESC (D
ToKNJ 漢 字 文 字 列 ToASCII

⇓ Cut up 6 bytes.



How should ";" be treated?

How about the rest of the sequence?

⟶ Break the rule of "SITI".

(Two illegal sequences occur in the above.)

9 Problems in Text Processing on Multibyte Chars.

- Encoding dependency.

  (Refer : Example A1)

- Necessity of managing Current Shift States.

  (Refer : Example A1)

- Difficulty/impracticability to concatinate/split strings. (Refer : Example A2, A3)

⇓

Wide Character Concept

10 Wide Character

Byte representation:

A	B	C	ESC	$	4	A	;	z	ESC	(	0	1

Extended Char Set:

Ⓐ Ⓑ Ⓒ 漢 字 ⓪ ①

Wide Character:

A	B	C	漢	字	0	1

The rule of "1 Char = 1 Object"
in wide char string.

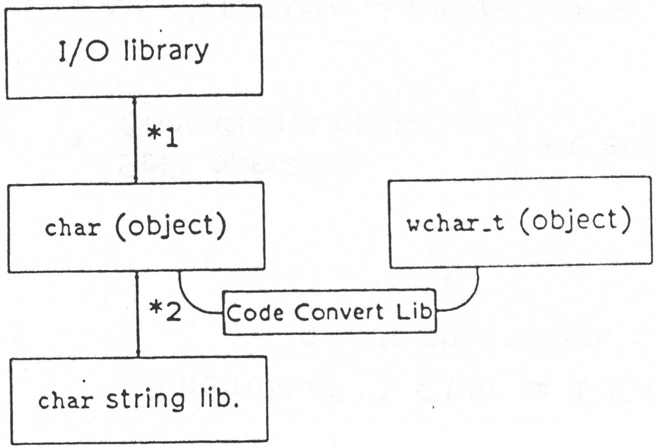
⇓

- Split operation
Concatinate operation } are ensured.

- No Shift State ⟶ Need not to manage it.

Character processing Model

11 ~~Current Status~~ of DIS9899


I/O library

*1

char (object)

*2

Code Convert Lib

wchar_t (object)

char string lib.

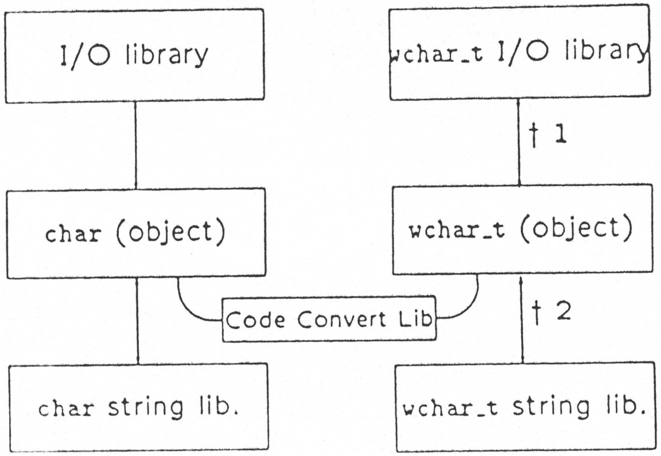
*1 Ensure byte data transparency.

*2 Concatination/split on char strings are not available.

MSE character Processing Model

11 ~~Current Status of DIS9899~~

~~with the MSE~~


I/O library

wchar_t I/O library

† 1

char (object)

wchar_t (object)

Code Convert Lib

† 2

char string lib.

wchar_t string lib.

†1 Implicit code converting.

†1 Ensure 1 Object = 1 Char.

12 Multibyte Support Extension (MSE)

[Approach]

Based on the specifications about multibyte handling in ISO/ANSI-C.

⇓

Enhance wide character handling functions for processing on wide char.

13 MSE (Continued)

~~[Guideline]~~
[Approach(detail)]

- Respect the specifications of multibyte handling in ISO/ANSI-C.

- Respect the spirit of encoding independency in ISO/ANSI-C.

  Discussion about shift encoding as in ISO-2022.

- Enhance wide char handling functions for processing on wide char.

- Prepare the Rationale for our specification to clarify the the reason and/or background of our decisions.

14 MSE (Continued)

[Basic Concept]

- Implicit code conversion.

  Programmers need not care about multibyte encoding.

- Establish the "1 Object = 1 Char" rule on even Large Char Set.

  It is possible to process text on wide. char as the same way as we have developed on char

  ⇓

The facilities that we introduce are:

- Wide Character/String Handling (isascii, strcpy ...)

- Wide Char I/O functions

  (~~printf~~, ~~scanf~~ ...)
  getwc putwc

15 MSE (Continued)

[C:ategorize ISO-C library functions]

Fi nctions which have char type arguments are cl: ssified as follows:

1. Wide char function corresponding to the char type functions are needed.
   (strcmp, strcpy, ...)
2. No new functions are needed but some modification about the argument/parameters is required.
   (printf, scanf, ...)
3. No new wide char functions are needed (its argument represents pathname).
   (rename, remove, fopen)
4. No new wide char functions are needed because of its rare usage.
   (atoi, atof, atol)

16 Current Status of DIS9899

char base		wchar_t base
<ctype.h>	char testing	
isalnum, isalpha, iscntrl, isdigit		
<stdio.h>	I/O	
fgetc, fputc, printf, scanf, sprintf		
<stdlib.h>	string convert	
strtod, strtol, strtoul		
<string.h>	string handling	
strcpy, strcat, strcoll, strxfrm, strcmp, strchr, strspn,		

⟶ Not sufficient for wide char processing.

16 Current Status of DIS9899 with MSE (Multibyte Support Extension)

char base		wchar_t base
<ctype.h>	char testing	<ctype.h>
isalnum, isalpha, iscntrl, isdigit		iswalnum, iswalpha, iscntrl, isdigit, set_wctype, is_wctype
<stdio.h>	I/O	<stdio.h>
fgetc, fputc, printf, scanf, sprintf		fgetwc, fputwc, (printf, scanf) wsprintf
<stdlib.h>	string convert	<stdlib.h>
strtod, strtol, strtoul		wcstod, wcstol, wcstoul
<string.h>	string handling	<string.h>
strcpy, strcat, strcoll, strxfrm, strcmp, strchr, strspn,		wcscpy, wcscat, wcscoll, wcsxfrm, wcscmp, wcschr, wcsspn,

⟶ ~~Not sufficient for wide char processing.~~

17 Multibyte Support Extension (Continued)

[Library Extension]

- To Enhance Wide Char Processing Facility

	WC testing
<ctype.h>	WC case mapping
	WC classification
	WC Input/Output
<stdio.h>	WC formatted I/O
<stdlib.h>	WC string conversion
<string.h>	WC string handling
<time.h>	WC time conversion (wcsftime)

18 Trend of Standardization about International features

- A lot of Standardization Bodies has begun to develop International features of Operating System.
  (X/Open, Posix, OSF, UII, Σ, etc.)
  → It tends to establish some different specifications.
  We would suffer from this difference.
  →The centre of such standardization activities are needed.

- They have decided to follow ~~determine to have respect for the libraries of~~ ISO/ANSI-C.
  →Now ISO/ANSI-C becomes one of the base ~~centres~~ of standardization of Open ~~Operating~~ Systems.

- Many ~~Several~~ implementations (ΣOS, AT&T MNLS, etc.) have appeared to prove our Proposal useful.

19 Conclusion

- We, the **ITSCJ** strongly ~~believe~~ propose that this MSE should be included in ISO-C as an Addendum.

  → If ISO-C would continue to have no such features, Some different specifications might be established.

- We propose the MSE to be adopted as a **New Work Item** of WG14.

- ~~When this proposal is accepted as a NWI,~~ The **ITSCJ** are ready to be the **Technical Editor** of this MSE.

A1 Example

- In case of no multibyte library functions ~~NOT available~~.

```
state = INITIAL;
while ((c = fgetc(fp)) != EOF) {
        switch (state) {
        case INITIAL:
                switch (c) {
                case ESC:
                        state = ESC;
                        break;
                        ...
                }
        }
}
```

⇓

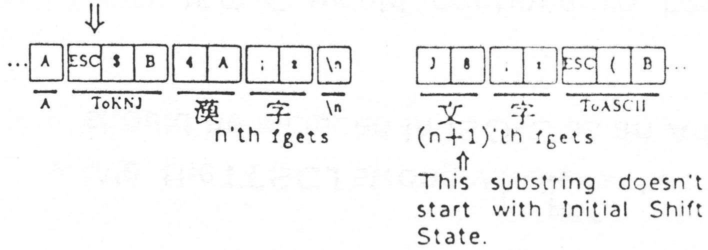
The program becomes encode dependent.

A2 Example(II)

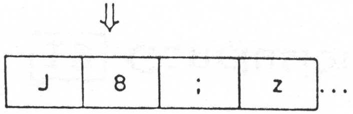
- In case of using mbstowcs.

```
while (fgets (buf, BUFMAX, fp) != NULL) {
        if (mbstowcs (wbuf, buf, WBUFMAX) == -1)
                error ("illegal code");
        }
        ....
}
```

fgets split the byte stream with '\n'



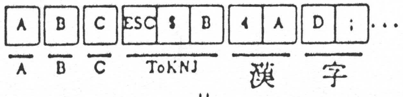
This substring doesn't start with Initial Shift State.



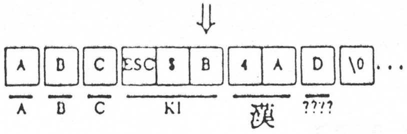
... as wide char string.

A3 Example(III)

- strtok with multibyte char string



```
strtok (p, ";");
```



The delimiter byte may be a part of multibyte char.

- strtok may break where it should not break
  (between a multibyte char)

- or the above behavior is prohibited in DIS9899?

Does DIS9899 tells strtok should recognize each multi-byte character according to the encoding rule?