国際化対応マルチバイト文字処理関数ライブラリの開発

林周志
rin@sfc.wide.ad.jp
環境情報学部 4 年 村井研究室
学籍番号:79054535

平成 6 年 3 月 1 日

Abstract

今日，ソフトウエアの国際化は大きな課題の一つのなっている．しかし世界中で使用されている文字コードにはさまざまなエンコーディングがある (日本語だけ考えてもロックシフトを用いた JIS コード，シングルシフトを用いた EUC，「すき間」を利用したシフト JIS，さらには近い将来 JIS 化されそうな Unicode がある)．これらすべてに対応するようにプログラミングすることは困難である．

この研究ではその解決策として、1 文字が 32 ビット固定長の 「ワイド文字」コードを設計し，ワイド文字コードとファイルで用いられている文字コード (マルチバイト文字コード) との変換を行なう入出力関数 (標準 C の stdio 関数に対応) とワイド文字を処理する関数 (標準 C の string,ctype 関数などに対応) を用意した．これによってプログラマからはどんな文字コードに対する操作もすべてワイド文字コードへの操作に抽象化される．

マルチバイト文字をワイド文字に変換して処理するライブラリ関数を用意するというのは一般的に用いられている方法である．しかし多くのライブラリでは対応しているマルチバイト文字コードは EUC などごく一部に限られている．とくにステートフルの文字コードに対応しているものはほとんどない．さらに，たいていのライブラリは特定のシステムに依存しており，システムごとに用意されている関数が異なるため，国際化対応アプリケーションの移植性を悪いものにしている．

この研究で作成したライブラリは現在使われているほとんどの文字コードに対応できる柔軟性がある．また特定のシステムに依存しないのでほとんどのシステムでこのライブラリを利用することができる．そのためこれを用いた国際化対応アプリケーションの移植性も高めることができる．

Abstract

The internationalization of software is one of the biggest problems we face today. Japan alone uses several different character codes such as JIS, EUC, SHIFT-JIS, and UNICODE (which will become JIS standard in the near future). Therefore, supporting all of the existing character codes is impossible at this point.

Solution for this problem, in my study, the "wide"character code with a fixed length of 32 bits was designed. Also prepared were I/O functions (corresponding to stdio function in the standard C library) that translate multibyte character codes used in files to/from wide character code, and wide character/string processing functions (that correspond to string and ctype functions in standard C library). By this, the operation for any character code is abstracted to the operation of wide character code.

Although preparing library functions that translate multibyte characters to/from wide characters and functions that process wide characters seems to be the general solution, most libraries only support a specific code such as EUC. Libraries that support stateful encoding are especially rare. Further more, almost all of these libraries depend on a specific system, therefore causing the prepared functions to differ and thus making the porting of internationalized applications more difficult.

This library has the flexibility to support almost any character code currently being used. Also, since this library does not depend on any specific system, it can be used on almost any system. By using this library, the portability of internationalized applications increases.

目 次

第 1 章 研究課題と成果 **4**
1.1 現状と課題 4
1.2 本研究での解決 4

第 2 章 国際化対応ライブラリについて **5**
2.1 ソフトウエアの国際化とは 5
2.1.1 地域化 5
2.1.2 多言語化 5
2.1.3 国際化 6
2.2 国際化アプリケーションのプログラミング 6
2.2.1 ロケールモデルの概要 6
2.2.2 ロケールモデルに従ったマルチバイト文字処理 7
2.3 既存の国際化対応文字処理関数ライブラリの実装 9
2.3.1 ライブラリの比較 10
2.3.2 既存の実装についてのまとめ 12

第 3 章 本ライブラリの設計 **17**
3.1 目標 17
3.2 ライブラリの設計 17
3.2.1 関数インタフェースの仕様 17
3.3 対応すべきマルチバイト文字コード 18
3.3.1 ステートレスのマルチバイトエンコーディング 18
3.3.2 ISO DIS10646/Unicode をもとにしたコード 18
3.3.3 ISO2022 に準拠したロックシフトエンコーディング 22
3.3.4 ロックシフトエンコーディングを扱う場合の問題 22
3.4 ワイド文字コードの選択 26
3.4.1 必要条件 26
3.4.2 候補 26
3.4.3 選択 29
3.5 ロケール情報の受け渡し、setlocale 関数の設計 29
3.5.1 ロケール名について 30

3.5.2 ロケール情報ファイル . . . 30
3.5.3 ロケール記述ファイルの内容 . . . 31
3.5.4 ロケール情報の受け渡し . . . 31
3.6 マルチバイト→ワイド文字変換関数の設計 . . . 31
3.6.1 変換アルゴリズムの設計 . . . 31
3.6.2 コードセットの判定 . . . 34
3.7 ワイド→マルチバイト文字変換関数の設計 . . . 36
3.8 ワイド文字入出力関数の設計 . . . 38
3.9 ワイド文字分類関数の設計 . . . 39
3.10 ワイド文字変換関数の設計 . . . 40
3.11 ワイド文字列関数群の設計 . . . 41

第 4 章 ライブラリの実装 42
4.1 内部でのロケール情報の保持 . . . 42
4.2 拡張マルチバイト→ワイド文字変換関数の実装 . . . 42
4.2.1 マルチバイト→ワイド文字変換アルゴリズム . . . 42
4.2.2 コードセット判別アルゴリズム . . . 45
4.3 拡張ワイド→マルチバイト文字変換関数の実装 . . . 49
4.4 ワイド文字 I/O 関数群の実装 . . . 50
4.5 ワイド文字分類関数群の実装 . . . 52
4.5.1 wctype 関数の実装 . . . 52
4.5.2 iswctype 関数の実装 . . . 53
4.6 ワイド文字変換関数群の実装 . . . 54
4.7 ワイド文字列関数群の実装 . . . 55
4.8 ロケール定義ファイル . . . 56

第 5 章 評価 61
5.1 対応できたマルチバイト文字コード . . . 61
5.2 対応できたシステム . . . 62
5.3 メモリ消費について . . . 62
5.4 処理速度について . . . 63
5.4.1 マルチバイト↔ワイド文字変換 . . . 63
5.4.2 ロケールデータのロード . . . 65

第 6 章 結論 66
6.1 このライブラリの利点 . . . 66
6.2 このライブラリの問題点 . . . 67

第 7 章 今後の課題 68
7.1 よりよいワイド文字の設計 . . . 68
7.2 アプリケーションへの適用 . . . 68

7.2.1 コンパイラへの適用 68
7.2.2 UNIX の文字処理ユーティリティの多言語化 68
7.2.3 高レベルライブラリの多言語化 68

参考文献 **70**

第 1 章

研究課題と成果

1.1　現状と課題

今日ではソフトウエア市場が世界的なものとなり、英語圏で開発されたソフトウエアもすぐに日本語その他の言語に対応できることが求められる。またインターネットの発達によって多言語のメッセージ交換も頻繁になり、個人の作成したフリーソフトウエアでさえすぐ世界中にばらまかれる。

このような現状からしてソフトウエアの国際化に対する要求が高まりつつあるといえる。しかし国際化対応ソフトウエアの開発を難しくするいくつかの問題がある。特に大きな問題と考えられるのは

> 世界中にはさまざまな文字コードが存在している。

ということである。1 文字 1 バイトという前提はすでに成り立たなくなっているし、1 文字を複数バイトで表現するコードにはいくつもエンコーディングが存在するのでそれらにすべて対応するのはかなり面倒である。それでどのようなコード体系でエンコードされていてもその違いを意識せずに文字を扱えるようなインタフェースを準備することが必要である。

そこでいくつかのシステムではマルチバイト文字処理関数ライブラリを提供している。しかし対応している文字コードが限定されており、用意されている関数にもかなりの違いがあるのが現状である。

1.2　本研究での解決

この研究では現在用いられている、また将来普及しそうなほとんどの文字コードを取り扱うことができる関数ライブラリを作成した。またこのライブラリはほとんどのシステムで利用することができ、このライブラリを利用した国際化対応アプリケーションの移植性を向上させることができる。

第 2 章

国際化対応ライブラリについて

この章は国際化アプリケーションのプログラミングとその支援のためのライブラリの現状についてのサーベイである。

2.1 ソフトウエアの国際化とは

まず最初にこの論文でのソフトウエアの 「国際化」の定義をしておこう。また 「国際化」に関係した用語として、 「地域化」、 「多言語化」という用語もある。これらもあわせて定義づけることにする。その後国際化アプリケーションの基本的なプログラミングについて概説する。

2.1.1 地域化

もともと英語圏用に開発されたソフトウエアを日本語対応にするという場合のようにソフトウエアをある地域の習慣に合わせてカスタマイズすることを指す。これには主にその地域で用いられている文字コードを扱えるようにすること、表示できるようにすることである。そのほかにもそのプログラムが出力するメッセージやメニューをその地域の言葉にすることやアイコンの絵をその地域の人々の感覚に合わせたものにすることなども含まれるだろう。身近な例としては xterm を日本語用に地域化した kterm、中国語用に地域化した cxterm などがある。

2.1.2 多言語化

複数の言語を同時に扱えるようにすること。これには普通 CompoundText や ISO10646/Unicode など新しい文字コードへの対応が関係している。

2.1.3 国際化

地域化を容易に行なうための機構が組み込まれていること。あるソフトウエアを地域化する場合はその地域への対応だけを考えてプログラミングする。そのため日本語対応のソフトウエアを作っても、それを中国語対応にしようという場合またプログラムを改造して、リコンパイルしなくてはならない。しかし国際化されたソフトウエアの場合、実行時に環境変数など外部からパラメータを与えるだけで日本語対応になったり、中国語対応になったりすることができる。文字コードへの対応、文字の表示ということだけについていえば、日本語 (例えば JIS コード)、中国語 (例えば GB2312) といった限られた地域での言語に対応するだけでなく、CompoundText、Unicode/ISO10646 などの多言語コードにも同様の手法で対応することができる。例えば xterm を国際化した exterm は「XLocale メニュー」があり、ここで "ja_JP.jis7" を選択すると日本語 7 ビット JIS コードに対応し、"zh_CN.big5" を選択すると中国語 big5 に対応、"wr_WR.ct" を選択すると CompoundText に対応できる。よってここでは国際化とは地域化と多言語化の両方を包含するものをさす (図 2.1)。

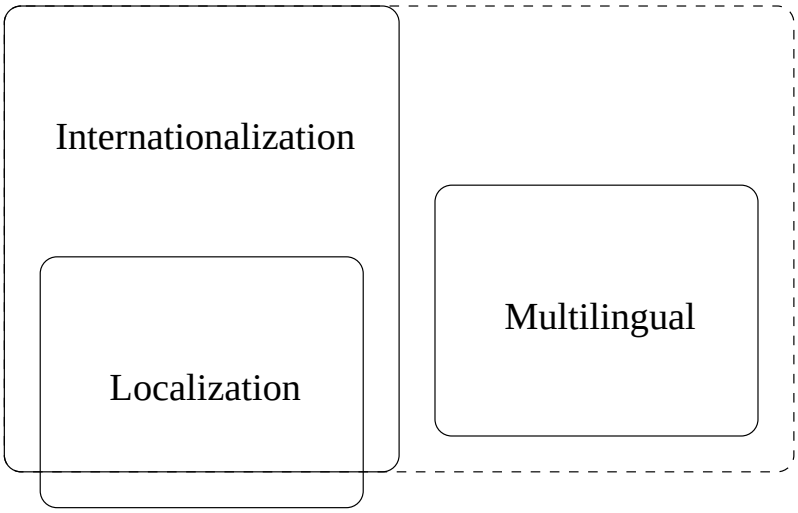


図 2.1: 地域化/他言語化/国際化の包含関係

2.2 国際化アプリケーションのプログラミング

2.2.1 ロケールモデルの概要

以上の定義より国際化されたソフトウエアは実行時に処理できる文字コード、表示する言語などが変更できなければならない。これを実現するためには一つの方法として、引数や環境変数で使用する言語を指定し、言語に依存する部分で

```
switch(language){
    case JAPANESE:
         日本語の場合の処理;
         break;
    case CHINESE:
         中国語の場合の処理;
          break;
    case US:
         英語の場合の処理;
         break;
}
```

のように言語ごとの処理を書くという方法も考えられる。しかしこれではプログラマの負担が大きく、ここに列挙されていない言語に新たに対応したいという場合、結局ソースを書き換えなければならなくなる。

そこで言語に依存する部分をプログラムから分離してファイルに保存し、プログラム実行時に指定された言語の情報をファイルからロードして、関数の動作を変えるという方法が考えられる。これが ANSI C で採用されたロケールモデルである。これに従った国際化アプリケーションはまず最初に

```
setlocale(LC_ALL, "");
```

という形式で setlocale 関数を呼ぶ。多くの実装では setlocale 関数は環境変数 LANG に設定されている言語名 (ロケール名) を取得して、規定されているディレクトリにおかれている言語依存情報のファイルをロードする。これ以降、言語に依存する処理を行なう関数はその情報を元に動作を変える。この方法だと言語の違いはかなりの程度ライブラリ関数によって吸収することができる。また新たに別の言語に対応しようという場合もプログラムの変更せず、その言語についての定義ファイルを用意してやればよい。ANSI C では言語に依存する部分をさらにいくつかのカテゴリ (ロケールカテゴリ) に分類している。ロケールはカテゴリごとに別々に設定することもできる。それぞれのカテゴリの意味と関係する関数について図 2.1にまとめる。

2.2.2 ロケールモデルに従ったマルチバイト文字処理

今後この論文では文字処理に関係した部分、つまりロケールカテゴリでは LC_CTYPE について主に議論する。そしてとくに 1 文字 1 バイトですむヨーロッパの言語ではなく、1 文字が可変長でマルチバイトで構成されるアジアの言語の文字コード (以下マルチバイト文字と呼ぶ) の処理について考慮する。

従来の C 言語標準ライブラリによる文字処理では 1 文字は 1 バイトであることを前提とし、バイトストリームから入出力 (stdio) 関数によってメモリにバイト列として読み込み、それを string 関数や ctype 関数で処理していた (図 2.2)。

表 2.1: ロケールの分類

ロケールカテゴリ	役割	関係する関数
LC_ALL	以下のすべてのカテゴリを示す	
LC_CTYPE	文字コード、文字の分類、操作	mbtowc,wctomb,is*,to*,printf など
LC_NUMERIC	数値の表記 (少数点、桁区切り)	printf,strtol など
LC_COLLATE	文字のソートする順序	strcoll,strxfrm など
LC_TIME	日付や時刻の表記方法	strftime
LC_MONETARY	金額の表記、貨幣単位	localeconv
LC_MESAGES	出力メッセージ	catopen, catgets など

注:LC_MESSAGES は XPG4 で定義されているが ANSI C にはない

マルチバイト文字のエンコーディング方法としてはさまざまなものが存在する。これらに個別に対応するのは難しい。それで ANSI C では 『ワイド文字」(wchar_t 型) というものが用意されている。まずストリームから読み込んだマルチバイト文字を現在のロケールにしたがって、1 文字が固定長でシフトステートのない処理しやすいワイド文字に変換する。そして文字処理はみなワイド文字に対して行なう。ストリームへ出力するときにはまたワイド文字からマルチバイト文字に変換する (図 2.3)。

ワイド文字にどのようなコード体系を用いるかは実装依存である (図 2.3 では ISO10646 UCS-4 を用いている)。マルチバイト文字とワイド文字の変換のために ANSI C では以下の 4 つの関数を用意している。

mbtowc()	マルチバイト文字をワイド文字へ変換
wctomb()	ワイド文字をマルチバイト文字へ変換
mbstowcs()	マルチバイト文字列をワイド文字列へ変換
wcstombs()	ワイド文字列をマルチバイト文字列へ変換

よって ANSI C の標準ライブラリの文字処理関数の階層は図 2.4のようになっている。しかし ANSI C ではこれら変換のための関数しか用意しておらず、ワイド文字に対する処理を行なう関数はない。またワイド文字を入出力するためにはいったんマルチバイト文字列との変換をして標準入出力関数を使わなければならない。このため ANSI C に用意されている関数群だけで国際化アプリケーションを作成するのはかなり面倒である。それで ANSI C 標準ライブラリを拡張した国際化対応文字処理関数ライブラリを用意する。これは以下のような機能を持った関数群である。

- ワイド文字 **I / O** 関数群... 標準ライブラリの stdio 関数のワイド文字対応版。入力関数はマルチバイト文字のストリームからバイト列を読み込んで内部でワイド文字に変換してメモリに格納する。また出力関数はワイド文字を内部でマルチバイト文字列に変換してストリームに書き出す。

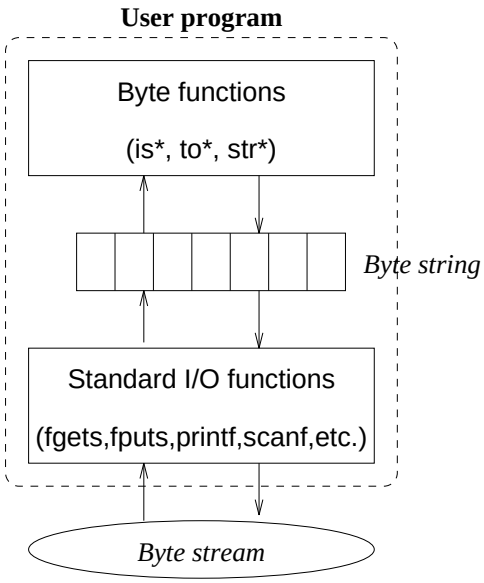


図 2.2: 伝統的な文字処理ライブラリ関数の階層

- ワイド文字処理関数群... 標準ライブラリの ctype, string 関数のワイド文字対応版。

これらによって標準ライブラリで ASCII 文字を処理する場合とほとんど変わらないインタフェースでマルチバイト文字の処理を行なうことができる (図 2.5)。

2.3 既存の国際化対応文字処理関数ライブラリの実装

上記のような機能を持ったライブラリを備えたシステムはすでにいくつか存在する。それらについて検討してみる。

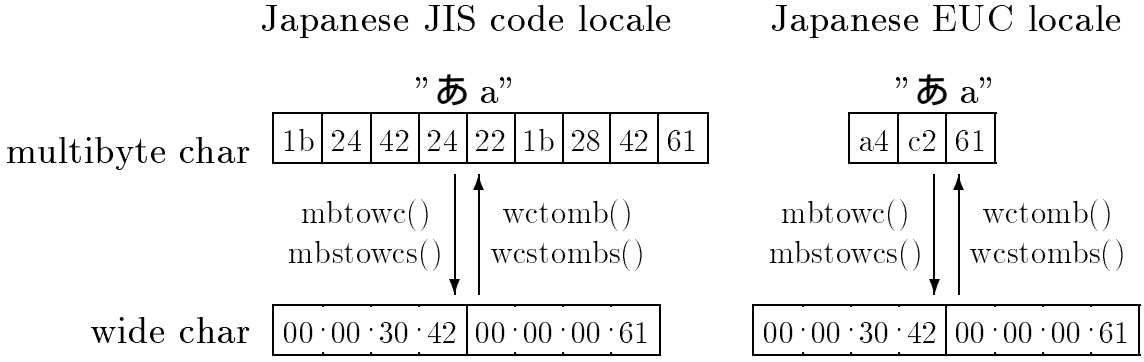


図 2.3: マルチバイト文字とワイド文字の変換

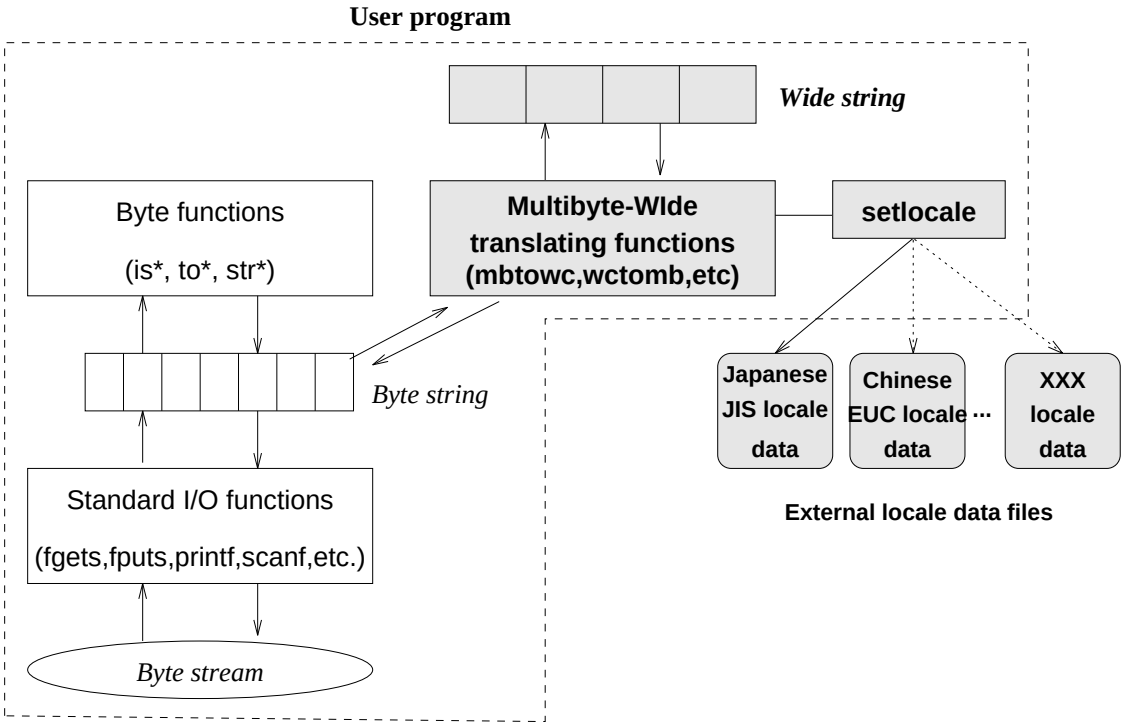


図 2.4: ANSI C での文字処理ライブラリ関数の階層

2.3.1 ライブラリの比較

最近の多くのシステムは独自の国際化対応文字処理関数ライブラリをもっている。実際にはここでいう国際化の定義にはあてはまらず、地域化または多言語化のレベルのものもあるが、システムごとの対比のために取り上げた。

Plan9[2]

ANSI C のロケールモデルには準拠していない。マルチバイト文字コードは UTF-2 のみでワイド文字コード (rune 型) は Unicode。よってここでいう 「国際化」にはあてはまらず、 「多言語化」対応といえる。マルチバイト↔ワイド文字の変換関数、いくつかの文字列処理関数、書式つき出力関数を用意している。

4.4BSD

マルチバイト文字コードとして各国語の EUC をサポート可能である。この場合はワイド文字コードとして EUC の変形で 16 ビット固定長のコードを用いる。またマルチバイト文字として UTF-2 もサポートしているが、この場合はワイド文字コードは Unicode になる。マルチバイト↔ワイド文字変換関数と Plan9 とよく似た文字列処理関数をもち、さらに文字分類関数 (isalpha,toupper など) がワイド文字対応になっている。

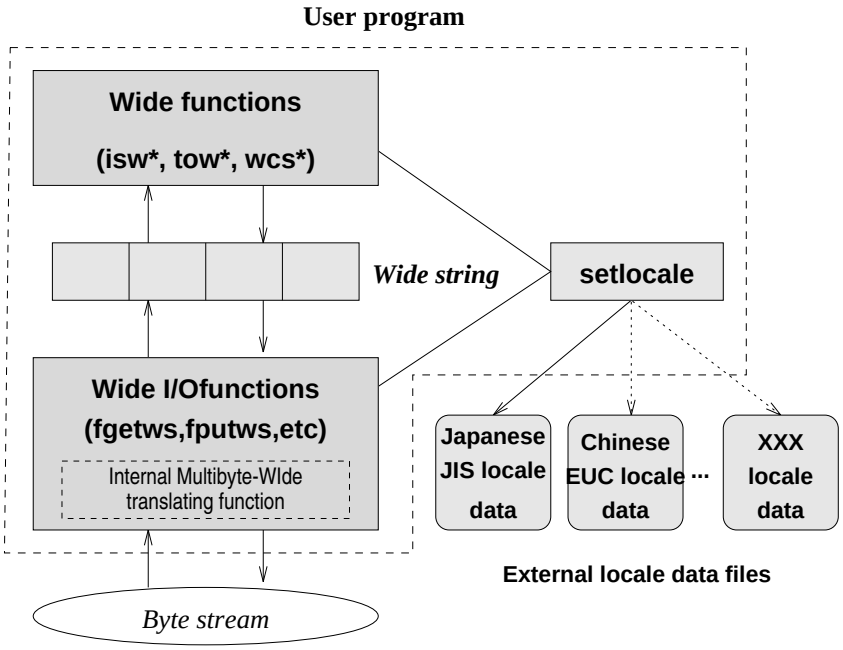


図 2.5: 国際化対応文字処理ライブラリ関数の階層

SunOS[4]

マルチバイト文字コードは EUC。ワイド文字コードは EUC の変形で 16 ビット固定長である。実際には日本語向けには JLE(Japanese Language Environment)、韓国語向けには KLE(Korean Language Environment) というふうに OS 自体が別パッケージになっており、「地域化」の域を越えていない。関数としては C の標準ライブラリのほとんどすべての関数のワイド文字対応版が用意されている。

SVR4.1 MNLS[5]

MNLS とは Multi-Nation Language Support の略である。マルチバイト文字コードは EUC。ワイド文字コードは 32bit の EUC 変形である。標準ライブラリのほとんどの関数のワイド文字対応版が用意されている。

HP-UX リリース 9.0 NLS[8]

NLS とは Native Language Support の略である。マルチバイト文字コードとして EUC と HP-15 という HP 独自の 2 バイトコードをサポートしている。ワイド文字コードについては不明。XPG4 の WPI(World-wide Portability Interface) に準拠したワイド文字対応関数を準備している。

それぞれのライブラリが対応可能なマルチバイト文字コードについて図 2.2に、各ライブラリで用意されている関数について図 2.3、2.4、2.5にまとめる。

表 2.2: 各ライブラリが対応可能なマルチバイト文字コード

	Plan9	4.4BSD	SunOS	SVR4	HP-UX
各国語 EUC		○	△	○	○
シフト JIS					
Big5					○
UTF-2	○	○			
ISO2022 準拠ロックシフト					

2.3.2 既存の実装についてのまとめ

対応している文字コードについて

どのライブラリも国際化というには対応しているマルチバイト文字コードが少ない。多くの「国際化」ライブラリ実装ではマルチバイト文字として EUC を用いて、各国語へ対応するというアプローチをとっている。たしかにマルチバイト文字を EUC であると限定し、ワイド文字は EUC を 1 文字固定長にしたものを用いれば変換が用意である。しかし現実には EUC はそれほど一般に使われているコード体系ではない。日本語ではシフト JIS や JIS コードも存在するし、中国語でも Big5 や GB2312 がある。とくに JIS コードなど ISO2022 に準拠したロックシフトを用いるエンコーディングに対応できるものはほとんどない。コンピュータをスタンドアロンで、または特定のシステムだけでネットワークを組んで利用しているのならばよいかもしれないが、実際にはその他のさまざまなシステムとのデータの交換は避けられないだろう。また EUC に基づいているのでは多言語対応することは難しい。

もう一つのアプローチは Plan9 や 4.4BSD にみられるように Unicode/ISO10646 をワイド文字として用い、マルチバイト文字はその変換形である UTF-2 を用いる方法である。これだと多言語対応が可能であるが、マルチバイト文字を限定しているという点で EUC を用いた実装とおなじ欠点を持つ。世界中のすべてのコンピュータで単一の文字コードが使われるようになれば理想的だが、その可能性は非常に薄いだろう。

用意されている関数について

HP-UX や SunOS などは十分な数の関数を用意しているといえる。しかしそのインタフェース (関数名や引数) にかなりの相違がある。多くの商用のシステムに附属しているライブラリは XPG(X/Open Portability Guide) に準拠していると唱っているがその第 3 版 (XPG3) までは国際化といっても 8 ビット文字、つまりヨーロッパの言語のことしか考慮しておらず、日本語、中国語、韓国語といったマルチバイト文字への対応はシステムごとにかなり相違ができてしまった。最新の X/Open Portability Guide は XPG4 であり、これでは World-wide Portability Interface としてマルチバイト文字への対応が考慮されている。しかしこれを実装しているシステムは HP-UX,OSF/1 な

どまだ少数である (SVR4.1 も XPG4 に合わせていると記述されているが図 2.3、2.4、2.5を見てもわかるとおり、HP-UX とはかなり違う)。それぞれのライブラリはライセンスなどの問題があり公開されていない (例外は 4.4BSD で、これのライブラリのロケール関係の部分はパブリックドメインとなって配布されている) のでシステム付属の国際化対応ライブラリを使って作成されたアプリケーションは移植性が非常に悪いものになってしまう。

表 2.3: 各ライブラリの用意している関数の比較 (入出力関数)

機能, 対応するバイト関数	Plan9	4.4BSD	SunOS JLE1.1	HP-UX9.0 NLS	SVR4.1 MNLS
関係するヘッダファイル					
	u.h	rune.h	widec.h euc.h wctype.h	wchar.h	widec.h sys/euc.h getwidth.h wctype.h
入出力関数					
fgetc			fgetwc	fgetwc	fgetwc
getc			getwc	getwc	getwc
getchar			getwchar	getwchar	getwchar
fgets			fgetws	fgetws	fgetws
gets			getws		getws
fputc			fputwc	fputwc	fputwc
putc			putwc	putwc	putwc
putchar			putwchar	putwchar	putwchar
fputs			fputws	fputws	fputws
puts			putws		putws
ungetc			ungetwc	ungetwc	ungetwc
書式つき入出力関数					
printf	print		printf	printf	printf
fprintf			fprintf	fprintf	fprintf
sprintf			sprintf, wsprintf	sprintf	sprintf
vprintf				vprintf	vprintf
vfprintf				vfprintf	vfprintf
vsprintf				vsprintf	vsprintf
scanf			scanf	scanf	scanf
fscanf			fscanf	fscanf	fscanf
sscanf			sscanf, wsscanf	sscanf	sscanf
vscanf				vscanf	vscanf
vfscanf				vfscanf	vfscanf
vsscanf				vsscanf	vsscanf
書式つき入出力関数の変換文字					
ワイド文字			%wc	%C	%wc
ワイド文字列	%S		%ws	%S	%ws

表 2.4: 各ライブラリの用意している関数の比較 (文字列処理関数)

機能, 対応するバイト関数	Plan9	4.4BSD	SunOS JLE1.1	HP-UX9.0 NLS	SVR4.1 MNLS
文字列処理関数					
index			windex		
rindex			wrindex		
strcat			wscat	wcscat	wscat
strchr	utfrune	mbrune	wschr	wcschr	wschr
strcmp			wscmp	wcscmp	wscmp
strcpy			wscpy	wcscpy	wscpy
strcspn			wscspn	wcscspn	wscspn
strlen	utflen		wslen	wcslen	wslen
strncat			wsncat	wcsncat	wcncat
strncmp			wsncmp	wcsncmp	wsncmp
strncpy			wsncpy	wcsncpy	wsncpy
strpbrk			wspbrk	wcspbrk	wspbrk
strrchr	utfrrue	mbrrune	wsrchr	strrchr	wsrchr
strspn			wsspn	wcsspn	wsspn
strstr	utfutf	mbmb		wcswcs	
strtok			wstok	wcstok	wstok
strcoll				wcscoll	
strftime				wcsxfrm	
strdup			wcdup		
表示幅を取得			wscol		getwidth
atof			watof		
strtod			wstod	wcstod	
atoi			watoi		
atol			watol		
strtol			wstol	wcstol	
strtoul				wcstoul	

表 2.5: 各ライブラリの用意している関数の比較 (文字分類 ・変換関数)

機能, 対応する バイト関数	Plan9	4.4BSD	SunOS JLE1.1	HP-UX9.0 NLS	SVR4.1 MNLS
ワイド文字分類 ・変換関数					
isalnum			iswalnum	iswalnum	iswalnum
isalpha			iswalpha	iswalpha	iswalpha
iscntrl			iswcntrl	iswcntrl	iswcntrl
isdigit			iswdigit	iswdigit	iswdigit
isgraph			iswgraph	iswgraph	iswgraph
islower			iswlower	iswlower	iswlower
isprint			iswprint	iswprint	iswprint
ispunct			iswpunct	iswpunct	iswpunct
isspace			iswspace	iswspace	iswspace
isupper			iswupper	iswupper	iswupper
isxdigit			iswxdigit	iswxdigit	iswxdigit
ASCII 文字の判定			iswascii		
英字の判定			isenglish		isenglish
表意文字の判定			isideogram		isideogram
表音文字の判定			isphonogram		isphonogram
特殊文字の判定			isspecial		isspecial
数字の判定			isnumber		isnumber
tolower			towlower	towlower	towlower
toupper			towupper	towupper	towupper

第 3 章

本ライブラリの設計

3.1 目標

これまで述べてきたような現状での問題点を解決するためには以下のような基本的要求を満たす必要がある。

1. 特定のマルチバイト文字コードを仮定せず、さまざまなエンコーディングにロケール情報のファイルを用意するだけで対応できること。
2. 標準 C ライブラリ関数とほとんど同様のインタフェースを提供すること。
3. このライブラリを用いたプログラムの移植性を高めるため、ライブラリ自体も特定のシステムに依存せず、移植性の高いものとすること。

3.2 ライブラリの設計

3.2.1 関数インタフェースの仕様

用意するべき関数は既存の実装と大きな違いがあるわけではない。全く独自に設計することもできるが、標準ライブラリのインタフェースとの互換性を重視するわけであるからそれほどオリジナリティが出せるわけではない。それでさまざまなところで議論されている国際化対応プログラミングインタフェース標準規格案の中でこれらの条件を満たしているものがあれば、その仕様にしたがったものを作成した方が配布する時にも受け入れられやすいだろう。そのような規格案として

- ISO C マルチバイト拡張仕様案 (MSE: Multibyte Support Extension)[12]
- Uniforum-X/Open Joint Internationalization Group(XoJIG) 案

がある。どちらについてもまだ完全に従った実装は存在しない。

MSE は日本の SC22/C 言語委員会が中心となって進めている ISO C(ISO/IEC 9899:1990) の追加規定案で、XPG4 の WPI もこれの古いドラフトをもとに設計されている。豊

富な関数が規定されており、最新の案ではロックシフトエンコーディングへの対応も考慮されている (ロックシフトエンコーディングへの対応に特別の注意が必要な理由は 3.3.4章で取り扱う)。

2 番めの XoJIG の案はかなり先進的なもので、「ロケールオブジェクト」という考え方を導入し、マルチスレッドのアプリケーションや分散アプリケーションでの国際化対応インタフェースについて考慮されている。

この研究のライブラリ関数インタフェースとしては MSE を採用することにした。XoJIG 案も非常に興味深いが、インタフェースが独特のものであること、まだそこまでの機能に対する需要は多くないと考えて却下した。しかし将来的にはぜひ検討してみたい。MSE には表 3.1、3.2、3.3のような ANSI C にはない新しい関数と型、ヘッダが定義されている。

3.3 対応すべきマルチバイト文字コード

「国際化」を唱うからには、現在世界中でよく用いられているほとんどすべての文字コードをサポートできるべきである。ヨーロッパ言語用の 1 バイトコード ISO8859 シリーズ以外に対応を考慮すべきコード体系をいくつかに分類してみる。

3.3.1 ステートレスのマルチバイトエンコーディング

これには

- 各国語 EUC
- シフト JIS
- Big-5

が含まれる。これらは 8 ビットの中で ASCII コードが用いていない部分、10 進数の値で言えば 128 から 255 を利用して、 1 文字のバイト数を識別できる。

3.3.2 ISO DIS10646/Unicode をもとにしたコード

最近注目されており、議論の的になることも多い ISO10646/Unicode(以下 Unicode)[9][11] であるがこれへの対応も考える必要がある。しかし Unicode は現在の C 言語の仕様にはなじまない。C 言語では文字はバイトの列からなり、値が 0 のバイトはヌルバイトで文字列の終端を表すことになっているが、Unicode は 1 文字 16 ビットであり、これを 2 バイトからなるマルチバイト文字であると考えるべきではない。例えば 'A' は Unicode では 16 進数で 0041 という値となるが、これでは文字列の途中でヌルバイトが入ってしまうことになる (当然逆に下位バイトが 0 となる文字もある)。また CPU アーキテクチャの違う計算機同士で Unicode のテキストを交換する場合はバイトオーダを考慮する必要もある。

表 3.1: MSE で規定されているヘッダ、型、関数–1

ヘッダ<iso646.h> 代用スペル	
マクロ	意味
and	&& の代用
and_eq	&= の代用
bitand	& の代用
bitor	| の代用
compl	~ の代用
not	! の代用
not_eq	!= の代用
or	|| の代用
or_eq	|= の代用
xor	^ の代用
xor_eq	^= の代用
ヘッダ<wctype.h> ワイド文字分類、変換関数群	
型	意味
wctrans_t	ロケール特有の文字変換を表現
wctype_t	ロケール特有の文字分類を表現
wint_t	WEOF を保持できる型
マクロ	意味
WEOF	end-of-file
関数	機能
int iswalnum(wint_t wc);	isalnum のワイド文字版
int iswalpha(wint_t wc);	isalpha のワイド文字版
int iswcntrl(wint_t wc);	iscntrl のワイド文字版
int iswdigit(wint_t wc);	isdigit のワイド文字版
int iswgraph(wint_t wc);	isgraph のワイド文字版
int iswlower(wint_t wc);	islower のワイド文字版
int iswprint(wint_t wc);	isprint のワイド文字版
int iswpunct(wint_t wc);	ispunct のワイド文字版
int iswspace(wint_t wc);	isspace のワイド文字版
int iswupper(wint_t wc);	isupper のワイド文字版
int iswxdigit(wint_t wc);	isxdigit のワイド文字版
wctype_t wctype(const char *property);	ワイド文字分類クラスの値を返す
int iswctype(wint_t wc, wctype_t desc);	拡張ワイド文字分類関数
wint_t towlower(wint_t wc);	tolower のワイド文字版
wint_t towupper(wint_t wc);	toupper のワイド文字版
wctrans_t wctrans(const char *property);	ワイド文字変換クラスの値を返す
wint_t towctrans(wint_t wc, desc);	拡張ワイド文字変換関数

表 3.2: MSE で規定されているヘッダ、型、関数–2

ヘッダ<wchar.h> 拡張マルチバイト/ワイド文字関数群	
型	意味
mbstate_t	変換ステートの保持
マクロ	意味
WCHAR_MAX	wchar_t の最大値
WCHAR_MIN	wchar_t の最小値
関数	機能
・ワイド文字入出力関数	
int fwprintf (FILE *stream, const wchar_t *format, ...);	fprintf のワイド文字版
int fwscanf (FILE *stream, const wchar_t *format, ...);	fscanf のワイド文字版
int wprintf (const wchar_t *format, ...);	printf のワイド文字版
int swscanf (const wchar_t *s, const wchar_t *format, ...);	sscanf のワイド文字版
int wscanf (const wchar_t *format, ...);	scanf のワイド文字版
int swprintf (wchar_t *s, size_t n, const wchar_t *format, ...);	sprintf のワイド文字版
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);	vfprintf のワイド文字版
int vwprintf (const wchar_t *format, va_list arg);	vprintf のワイド文字版
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list);	vsprintf のワイド文字版
wint_t fgetwc (FILE *stream);	fgetc のワイド文字版
wchar_t *fgetws (wchar_t *s, int n, FILE *stream);	fgets のワイド文字版
wint_t fputwc (wint_t c, FILE *stream);	fputc のワイド文字版
int fputws (const wchar_t *s, FILE *stream);	fputs のワイド文字版
wint_t getwc (FILE *stream);	getc のワイド文字版
wint_t getwchar(void);	getchar のワイド文字版
wint_t putwc (wint_t c, FILE *stream);	putc のワイド文字版
wint_t putwchar(wint_t c);	putchar のワイド文字版
wint_t ungetwc (wint_t c, FILE *stream);	ungetc のワイド文字版
int fwide (FILE *stream, int mode);	ストリームのオリエンテーション変更
・ワイド文字列数値変換関数	
double wcstod (const wchar_t *nptr, wchar_t **endptr);	strtod のワイド文字版
long int wcstol (const wchar_t *nptr, wchar_t **endptr, int base);	strtol のワイド文字版
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);	strtoul のワイド文字版

表 3.3: MSE で規定されているヘッダ、型、関数–3

関数	機能
・ワイド文字列関数	
wchar_t *wcscpy (wchar_t *s1, const wchar_t *s2);	strcpy のワイド文字版
wchar_t *wcsncpy (wchar_t *s1, const wchar_t *s2, size_t n);	strncpy のワイド文字版
wchar_t *wcscat (wchar_t *s1, const wchar_t *s2);	strcat のワイド文字版
wchar_t *wcsncat (wchar_t *s1, const wchar_t *s2, size_t n);	strncat のワイド文字版
int wcscmp (const wchar_t *s1, const wchar_t *s2);	strcmp のワイド文字版
int wcscoll (const wchar_t *s1, const wchar_t *s2);	strcoll のワイド文字版
int wcsncmp (const wchar_t *s1, const wchar_t *s2, size_t n);	strncmp のワイド文字版
size_t wcsxfrm (const wchar_t *s1, const wchar_t *s2, size_t n);	strxfrm のワイド文字版
wchar_t *wcschr (const wchar_t *s, wint_t c);	strchr のワイド文字版
size_t wcscspn (const wchar_t *s1, const wchar_t *s2);	strcspn のワイド文字版
wchar_t *wcspbrk (const wchar_t *s1, const wchar_t *s2);	strpbrk のワイド文字版
wchar_t *wcsrchr (const wchar_t *s, wint_t c);	strrchr のワイド文字版
size_t wcsspn (const wchar_t *s1, const wchar_t *s2);	strspn のワイド文字版
wchar_t *wcsstr (const wchar_t *s1, const wchar_t *s2);	strstr のワイド文字版
wchar_t *wcstok (wchar_t *s1, const wchar_t *s2, wchar_t **ptr);	strtok のワイド文字版
size_t wcslen (const wchar_t *s);	strlen のワイド文字版
size_t *wmemchr (const wchar_t *s, wchar_t , size_t n);	memchr のワイド文字版
int wmemcmp (const wchar_t *s1, const wchar_t *s2, size_t n);	memcmp のワイド文字版
wchar_t *wmemcpy (wchar_t *s1, const wchar_t *s2, size_t n);	memcpy のワイド文字版
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);	memmove のワイド文字版
wchar_t *wmemset (wchar_t *s, wchar_t, size_t n);	memset のワイド文字版
size_t wcsftime(wchar_t *s, size_t max, const wchar_t *format, const struct tm *timeptr);	strftime のワイド文字版
・拡張マルチバイト/ワイド文字変換関数	
wint_t btowc (int c);	1 バイトをワイド文字へ変換
int wctob (wint_t c);	ワイド文字を 1 バイト文字へ変換
int mbsinit (const mbstate_t *ps);	ps が初期状態か調べる
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);	mblen の “restartable” 版
size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);	mbtowc の “restartable” 版
size_t wcrtomb (char *s, wchar_t wc, mbstate_t *ps);	mctomb の “restartable” 版
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);	mbstowcs の “restartable” 版
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);	wcstombs の “restartable” 版

それで Unicode をそのままマルチバイト文字として扱うことはできないが、その変換形式である UTF をマルチバイト文字として扱えるようにする。UTF とは UCS(Universal multiple-octet Coded Character Set) Transformation Format の略で、基本的な考え方は Unicode を ASCII 互換のマルチバイト文字コード、つまり Unicode の ASCII コードに対応する部分 (制御文字を含む) は 1 バイトの ASCII コードに変換し、それ以外の部分はヌル文字、制御文字を含まない形式のマルチバイト文字に変換するというものである。UTF にはいくつかの形式がある。いずれもステートレスエンコーディングである。

- **UTF...** Unicode の付属書に規定されている方式
- **UTF-2...** X/Open が提案した。FSS(File System Safe)-UTF とも呼ばれる。変換されたコードが 1 文字複数バイトになる場合はかならず各バイトの 8 ビットめが 1 になるのでマルチバイト文字中のバイトで '/' に対応する ASCII コードが出てくることはない。また Unicode の付属書の方法より変換が容易である。
- **I-UTF[14]...** 東工大の太田昌孝氏が提案した方式。UTF-2 をさらに発展させ、マルチバイト文字中に ASCII の '/' にあたるコードだけが出てこないようにして、必要なバイト数を減らしている。
- **UTF-7...** まだ具体的な変換方法は決まっていないようだが、各バイトの 7 ビット分だけを使い、最上位ビットを使わないことによって通信に使いやすいようにしている。

3.3.3 ISO2022 に準拠したロックシフトエンコーディング

エスケープ文字とそれに続くエスケープシーケンスによってコードセットの切替を行なう。

- **JIS コード**
- **CompoundText**
- **ISO-2022-JP-2**
- **ISO-2022-INT-1**

などが含まれる。

3.3.4 ロックシフトエンコーディングを扱う場合の問題

このタイプのコードはステートをもつため、マルチバイト↔ワイド文字変換を行なう関数は内部で現在のステートを保存しなければならない。ANSI C の規格でも mbtowc,wctomb 関数は内部でステートを保存するように考慮されている。しかし実

```
 1: #include <stdlib.h>
 2: #include <stdio.h>
 3: #include <locale.h>

 4: main()
 5: {
 6:     FILE *fp1, *fp2;
 7:     char mb1[BUFSIZ], mb2[BUFSIZ];
 8:     wchar_t wc1, wc2;

 9:     setlocale(LC_ALL, ""); /* ロケールの設定 */

10:     fp1 = fopen("file1", "r");
11:     fp2 = fopen("file2", "r");

12:     fgets(mb1, BUFSIZ, fp1);
13:     fgets(mb2, BUFSIZ, fp2);

14:     mbtowc(&wc1, mb1, MB_CUR_MAX);
15:     mbtowc(&wc2, mb2, MB_CUR_MAX);
16: }
```

図 3.1: 複数のバイト列をワイド文字に変換するプログラム

際には単一のステートを保存するだけでは不十分である。ごく単純化した例だが図 3.1 のようなプログラムを考えてみる。

プログラム実行前に

```
setlocale LANG ja_JP.jis7
```

などしてステートフルの文字コードを用いるロケールにセットされているとする。mbtowc 関数はマルチバイト文字をワイド文字に変換する関数で

#include <stdlib.h>
int mbtowc(wchar_t *$wptr$**, const char ***ptr**, size_t** n**);**

ptr の指す n バイト以内をワイド文字に変換し、$wptr$ へ書き込む。**MB_CUR_MAX** は stdlib.h に定義されているマクロで、現在のロケールで 1 マルチバイト文字を表現するのに必要な最大バイト数を表す。 file1 の先頭の行は、

あいう (すべて ASCII で表すと “esc $ B $ " $ $ $ & esc (B”)

file2 の先頭の行は、

abc

だったとする。それぞれ mb1,mb2 に読み込まれる。14 行目で mb1 の最初の 1 文字分 (この場合 5 バイト) がワイド文字に変換されると同時に mbtowc の内部ステートは 1 文字 2 バイトの JISX0208 であることを示すものに切り替わる。この状態で 15 行目が実行されると、mbtowc は現在のステートにしたがって mb2 から 2 バイトをとりだし、それを JISX0208 の文字と解釈してワイド文字に変換しようとするため正しく変換されなくなってしまう (ちなみに 'ab' つまり 16 進で 6162 は JISX0208 では「猴」である)。

それできちんとステートフルな文字コードに対応するためにはステートを文字列ごと (入出力関数であればストリームごと) に管理しなければならない。MSE ではこれを実現するため、現在のステートを保存するために mbstate_t という型を用意している。そして ANSI C のマルチバイト/ワイド文字変換関数 mblen, mbtowc, wctomb, mbstowcs, wcstombs を拡張し、それぞれ mbrlen, mbrtowc, wcrtomb, mbsrtowcs, wcsrtombs という関数を用意している ('r' は 'restartable' の略である)。ANSI C 標準のマルチバイト/ワイド文字変換関数と MSE の拡張マルチバイト/ワイド文字変換関数の対応は以下のようになっている。

- 1 マルチバイト文字のバイト数を返す

  ANSI C: mblen(const char *mb, int n);
  MSE: mbrlen(const char *mb, size_t n, mbstate_t *ps);

- 1 マルチバイト文字をワイド文字に変換する

  ANSI C: mbtowc(wchar_t *wc, const char *mb, size_t n);
  MSE: mbrtowc(wchar_t *wc, const char *mb, size_t n, mbstate_t *ps);

- マルチバイト文字列をワイド文字列に変換する

  ANSI C: mbstowcs(wchar_t *wcs, const char *mbs, size_t n);
  MSE: mbsrtowcs(wchar_t *wcs, const char **mbs, size_t n, mbstate_t *ps);

- 1 ワイド文字をマルチバイト文字に変換する

  ANSI C: wcstombs(char *mbs, const wchar_t *wcs, size_t n);
  MSE: wcsrtombs(char *mbs, const wchar_t **wcs, size_t n, mbstate_t *ps);

- ワイド文字列をマルチバイト文字列に変換する

  ANSI C: wctomb(char *mb, wchar_t wc);

MSE: wcrtomb(char *mb, wchar_t wc, mbstate_t *ps);

ANSI C ではこれらの関数内部でステートを保存するようになっていたところが、MSE ではそれぞれの関数の最後の引数に mbstate_t 型の変数のアドレスを渡して、そこに変換終了時のステートを書き込む。それで変換される文字列へのポインタとステートを保存する変数とを組みで指定してやれば複数の文字列を交互に変換したとしても正しく変換することができる。また文字列変換関数 (mbsrtowcs, wcsrtombs) では第 2 引数で指定される被変換文字列の途中で変換がいったん終了した場合はその位置を保存することができるので、後でその続きから変換を再開することができる。またワイド文字入出力関数でもオープンしたファイルポインタとそのステートを組みで保存する必要がある。図 3.1のプログラムを MSE の拡張マルチバイト/ワイド文字処理関数を使って書き直すと図 3.2のようになる。

```
 1: #include <wchar.h>
 2: #include <stdio.h>
 3: #include <locale.h>

 4: main()
 5: {
 6:     FILE *fp1, *fp2;
 7:     char mb1[BUFSIZ], mb2[BUFSIZ];
 8:     wchar_t wc1, wc2;
 9:     mbstate_t st1, st2;

10:     setlocale(LC_ALL, "");

11:     fp1 = fopen("file1", "r");
12:     fp2 = fopen("file2", "r");

13:     fgets(mb1, BUFSIZ, fp1);
14:     fgets(mb2, BUFSIZ, fp2);

15:     mbrtowc(&wc1, mb1, MB_CUR_MAX, &st1);
16:     mbrtowc(&wc2, mb2, MB_CUR_MAX, &st2);
17: }
```

図 3.2: 複数のバイト列をワイド文字に変換するプログラム–MSE 版

3.4 ワイド文字コードの選択

3.4.1 必要条件

実際の処理に使われるワイド文字コードは最低限

- 1 文字が固定長
- ステートレス

である必要がある。またマルチバイト文字との変換が容易であればそれにこしたことはない。しかし UTF など Unicode 系と EUC などそれぞれの国内の標準規格を元にしたコード系とではとくに日本語、中国語、韓国語についてはコードの割り当て方が全く異なるため、すべてのマルチバイト文字コードとの変換が容易なワイド文字コードを設計することは不可能である。それでワイド文字の設計方針として以下の 2 通りが考えられる。

1. ワイド文字コードは複数のコーディングシステムを切替えられるようにし、ロケールによってマルチバイト文字と変換しやすいコーディングを使う。
2. すべてのマルチバイト文字コードに含まれる文字を含んだ単一のワイド文字コードを用いる。

前者はマルチバイト文字コードとして UTF-2 を用いるロケールではワイド文字コードとして Unicode を用い、EUC が用いられるロケールの時はワイド文字コードとしては SVR4 MNLS で用いられているような 32 ビット (もしくは 16 ビット) 固定の EUC の変形を用いるといった方法である。これだと、1) そのプログラムの中で setlocale は一回しか呼ばれない、つまりプログラムのロケールが途中で切り替わることはない。2) プログラマはワイド文字コードを直接操作することはなく、用意されたワイド文字処理関数を通してのみ操作する。ということが前提となる。しかし現実には、エディタなどではファイルを読む時とファイルに書き込む時とでロケールを変えて文字コードの変換を行なうようなプログラムも書かれるだろうし (図 3.3参照)、ワイド文字を直接操作することも十分考えられる。

よって本研究のライブラリでは後者の方針でワイド文字コードを選択する。それでワイド文字コードの必要条件としてさらに次の点が加わる。

- そのコードに含まれる文字集合は対応予定のすべてのマルチバイト文字コードのスーパセットでなければならない

3.4.2 候補

以上のような条件にかなり近いと思われる既存の文字コードとしては以下のようなものがある。

```
/*
    使い方: a.out [入力のロケール] [出力のロケール]
*/
#include <locale.h>
#include <wchar.h>
#include <stdio.h>
#inlcude <stdlib.h>

main(int argc, char *argv[])
{
    wchar_t wcs[BUFSIZ];

    if(argc != 3){
        fprintf(stderr, "usage: %s [input stream locale] [output stream locale]\n",
                argv[0]);
        exit(1);
    }
    setlocale(LC_CTYPE, argv[1]); /*カテゴリ LC_CTYPE を入力のロケールにセッ
ト*/
    while(fgetws(wcs, BUFSIZ, stdin) != NULL){
        setlocale(LC_CTYPE, argv[2]);/*カテゴリ LC_CTYPE を出力のロケールに
セット*/
        fputws(wcs, stdout);
        setlocale(LC_CTYPE, argv[1]);/*カテゴリ LC_CTYPE を入力のロケールに
セット*/
    }
}
```

図 3.3: ロケール変更とワイド文字入出力関数を用いた簡単なコード変換プログラム

ISO DIS10646-1.0[10]

ISO10646 の最初のドラフト。1 文字 32 ビット長で、既存の文字コードの規格 (ASCII、日本の JISX0208、JISX0212、中国の GB2312、韓国の KSC5601 など) との変換も容易である。しかし後述する最近の ISO DIS10646-1.2 ではまったくコード体系が異なっており、現在では廃案となっている。

ISO DIS10646-1.2BMP/Unicode[9][11]

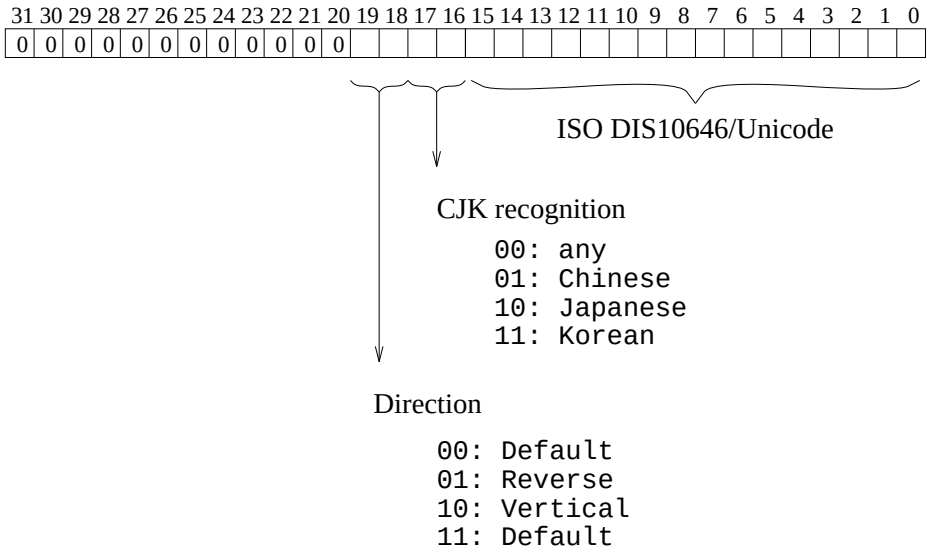
最新の ISO DIS10646BMP(Basic Multilingual Plane) と Unicode はほとんど同一のものになっている。1 文字 16 ビットで、ASCII や ISO8859 との変換はそれほどむずかしくないが、日中韓の漢字が統合されている (CJK Unify という)。当然コード順は JISX0208, 0212, GB2312, KSC5601 などとは全く異なるので EUC や JIS などとの変換が複雑になるだけでなく、そのコードで表現されている漢字が日本のものか中国のものか韓国のものか判別できない。それで日本語と中国語が混在したテキストなどを表現することがむずかしい。またヨーロッパの文字に関しても、 Non-spacing mark などとよばれるコードが存在し、1 文字が必ずしも固定長といえない面がある。例えば 'ȧ'、'ą̃' はそれぞれ以下のようなコードで表される。

$$\dot{\text{a}} \quad \text{a}(0\text{x}0061) + \dot{} \ (0\text{x}0307)$$

$$\tilde{\dot{\text{a}}} \quad \text{a}(0\text{x}0061) + \underset{\cdot}{} (0\text{x}0323) + \tilde{} \ (0\text{x}0303)$$

ICODE[14]

東工大の太田昌孝氏が提案。ISO DIS10646/Unicode に CJK 識別と方向性の識別のためのビットを追加した以下のような 32 ビット表現となっている。



それぞれの特徴をまとめると以下のようになる。

	ISO DIS10646-1.0	ISO DIS10646-1.2/Unicode	ICODE
ビット長	32	16	32
他のコードとの互換性	○	×	×
CJK 判別	○	×	○
non-spacing mark	なし	あり	あり

3.4.3 選択

どのコードもそれほど満足のいくものではない。ISO DIS10646-1.0 は EUC や CompoundText との変換がしやすいという点ではかなり魅力的だが、UTF との変換は複雑になる。また仕様書を入手することができなかった。ISO DIS10646-1.2/Unicode は CJK の識別ができないのが大きな問題である。マルチバイト文字コードとして CompoundText や ISO-2022-INT を使った場合、これらのコードは CJK は識別ができるので、ワイド文字に変換した時にその識別情報が落ちてしまってはならない。それで結局ワイド文字コードには **ICODE** を採用することにした。これにも non-spacing mark の問題があるがそれほど頻繁に用いられることはないと考えて目をつぶることにした。将来的にはもっと理想的なワイド文字コードを設計したい。それでマルチバイト ・ワイド文字変換関数はワイド文字コードのコード体系にできるだけ依存しないように作成することにした。(つまりコードの特徴を利用して多少でもスピードをあげようといったことを考えないようにした。)

これでワイド文字は 32 ビット長と決まったので wchar_t は unsigned long に型定義される。また wint_t も unsigned long に型定義する。

3.5 ロケール情報の受け渡し、setlocale 関数の設計

以上までで関数は MSE に準拠すること、ワイド文字は 32 ビット長で icode にならったものとすること、マルチバイト文字としてはロックシフトエンコーディングのものまで考慮することを決定した。これからはこの基本仕様をもとに各関数の具体的な設計を行なう。

表 3.1,3.2,3.3にはでてこないが、setlocale 関数もこのライブラリの関数用に作り直さなければならない。char *setlocale(int category, const char *locale) の仕様は以下のとおりである。

- 引数 category には新たにロケールをセットする対象になるロケールカテゴリの値 (マクロ定義されている) が入る。ロケールカテゴリとしては ANSI C では表 2.1にあるようなものが用意されているが、このライブラリで有効なのは LC_CTYPE と LC_ALL(実際に設定されるのは LC_CTYPE のみ) だけである。
- 引数 locale
  - NULL のとき... 現在設定されているロケールの名前を返す

  - ""のとき... 環境変数 LC_ALL, LC_CTYPE もしくは LANG(この順に優先順位が高い) に設定されたロケールにセット
  - ロケール名のとき... そのロケールにセット

setlocale 関数は指定されたロケールの情報ファイルを読み込んでその情報をロケールの影響を受ける関数に渡す。それで

- ロケール名はどうするか
- ロケール記述ファイルはどこにおくか
- ロケール記述ファイルには何を書くか
- ロケール情報をどのように他の関数に受け渡すか

を決める必要がある。

3.5.1 ロケール名について

ロケール名の付け方には明確な決まりはない。XPG4 では

language_territory.codeset

という付け方を提案している。これに従うと例えば日本語で文字コードとして EUC を使うロケールは **"ja_JP.eucJP"** などとする。多言語コードの場合も **"ja_JP.CTEXT"** などとすれば文字コードは CompoundText を用い、(このライブラリでは考慮しないが) メッセージ、時刻、日付の表記方法は日本語の習慣に合わせる (つまりカテゴリ LC_MESSAGES や LC_TIME などに対する指定) ということを表現できる。

3.5.2 ロケール情報ファイル

多くの実装ではロケール情報を

`/usr/lib/locale/{ロケール名}/{カテゴリ名}`

というファイルに保存している。例えばロケール ja_JP.eucJP のカテゴリ LC_CTYPE に関する情報であれば

`/usr/lib/locale/ja_JP.eucJP/LC_CTYPE`

におくことになる。また環境変数でパスを指定することもできる。このライブラリでもこれに倣って

`/usr/local/lib/locale/{ロケール名}/{カテゴリ名}`

をデフォルトのデフォルトのロケール記述ファイルパスとする。また環境変数でパスを指定できるようにする。

3.5.3 ロケール記述ファイルの内容

多くのマルチバイト文字コードに対応するため、文字コードに依存する情報はできるだけ LC_CTYPE ファイルに追い出し、ハードコーディングすることがないようにしたい。文字コードに依存する情報は大きく分けて

- マルチバイト↔ワイド文字変換に関する情報 (mbrtowc, wcrtomb などのマルチバイト/ワイド文字変換関数、 fgetwc, fputwc などのワイド文字 I/O 関数の動作に影響を与える)
- 文字分類に関する情報 (iswalpha,iswdigit などのワイド文字分類関数の動作に影響を与える)
- 文字変換に関する情報 (towupper, towlower などのワイド文字変換関数の動作に影響を与える)

がある。

3.5.4 ロケール情報の受け渡し

setlocale でロードされたロケール記述ファイルの内容によって各ライブラリ関数の動作が変わる。それで関数間でグローバルな構造体にこの情報を保持することにする。setlocale 関数だけがこの構造体を書き換える。こうすれば他の関数は呼び出されるたびに現在のロケールを確かめたりする必要はなくなる。

3.6 マルチバイト→ワイド文字変換関数の設計

これに関係するのは拡張マルチバイト/ワイド文字変換関数

```
mbrtowc    mbsrtowcs  mbrlen     btowc
```

の各関数である。これらは現在のロケールに従ってマルチバイト文字からワイド文字への変換を行なう。

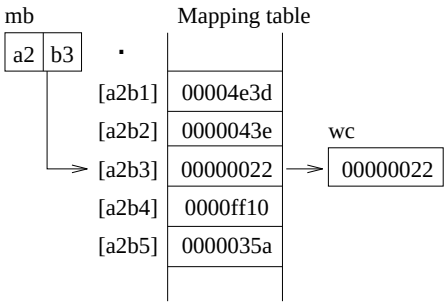
3.6.1 変換アルゴリズムの設計

変換アルゴリズムとしては大きく分けて

1. 一定の演算やビット操作によって変換する方法
2. マルチバイト文字とワイド文字とのマッピングテーブルによって変換する方法

が考えられる (図 3.4)。このライブラリの場合ワイド文字コードが ICODE のため EUC などとはコードの並びが全く異なる。そのため 1. の演算による方法では効率良く変換することができない。それで 2. のマッピングテーブルによる方法を用いることになる。

```
wchar_t mb2wc(mb)
    register char *mb;
{
    return(((wchar_t)(*mb << 8)
          + *(mb + 1)) | 0x8080);
}
```



演算による方法　　マッピングテーブルによる方法

図 3.4: マルチバイト→ワイド文字変換アルゴリズムの例

マッピングテーブルを用いる場合、メモリ消費が問題となる。2 バイトコードの値をそのままテーブルのエントリ (つまり配列の添字) として与えるとすると、$2^{16} = 65536$ エントリが必要になる。ワイド文字 (wchar_t) は 4 バイト長なので、テーブル全体の大きさは 65536×4 バイト=256k バイトとなる (後述するがワイド→マルチバイト変換テーブルも必要であることを忘れてはならない)。さらにマルチバイト文字が 3 バイトコードの場合は 64M バイト、4 バイトコードなら 4G バイトとなる。ここまでいくととても現実的な値ではなくなる。しかし実際にはコードのすべての部分が使われているわけではない。例えば日本語 EUC コードは最大 3 バイトであるが、文字が割り当てられている部分を各バイトをそれぞれ XYZ 軸に割り当てて 3 次元の空間として表現するとほとんどが空きであることが分かる (図 3.5)。

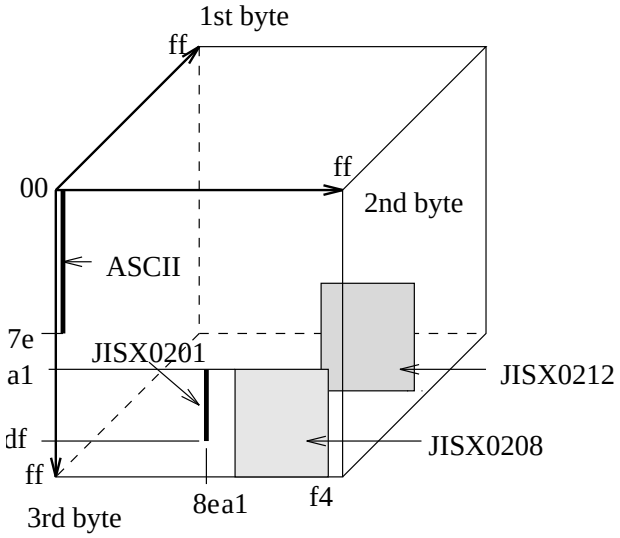


図 3.5: 日本語 EUC のコード割り当て空間

それでマッピングテーブルは文字の割り当てられている範囲ごとに作成することにする。ここでは便宜上ひとつのマッピングテーブルに割り当てられたマルチバイト文字コードの範囲のことを 「コードセット」と呼ぶことにする。これは UTF 以外の場合はそれぞれの国の標準規格の文字集合と対応する。例えば日本語 EUC の場合、コードセットの割り当ては以下のようになる。

コードセット	バイト数	第 1 バイトの範囲	第 2 バイトの範囲	第 3 バイトの範囲
0(ASCII)	1	0x00 – 0x7e		
1(JISX0208)	2	0xa1 – 0xf4	0xa1 – 0xfe	
2(JISX0201)	1	0x8e – 0x8e	0xa1 – 0xdf	
3(JISX0212)	3	0x8f – 0x8f	0xa1 – 0xed	0xa1 – 0xfe

こうすると変換を行なう前にそのマルチバイト文字コードがコードセットの範囲に入っているかどうかのチェックと若干の演算が必要になるが、テーブルのサイズは

コードセット 0	$(0x7e - 0x00 + 1) \times 4 = 508$	バイト
コードセット 1	$(0xf4 - 0xa1 + 1) \times (0xfe - 0xa1 + 1) \times 4 = 31588$	バイト
コードセット 2	$(0xdf - 0xa1 + 1) \times 4 = 256$	バイト
コードセット 3	$(0xed - 0xa1 + 1) \times (0xfe - 0xa1 + 1) \times 4 = 28952$	バイト
合計	61304	バイト

となる。

多言語コードで日中韓の文字が個別に入っている CompoundText の場合でも大まかに見積もって

コードセット	テーブルのサイズ
0(ASCII)	$94 \times 4 = 376$
1(JISX0201-1976 カタカナ)	$94 \times 4 = 376$
2(JISX0201-1976 ローマ字)	$94 \times 4 = 376$
3(Latin-1)	$96 \times 4 = 384$
4(Latin-2)	$96 \times 4 = 384$
5(Latin-3)	$96 \times 4 = 384$
6(Latin-4)	$96 \times 4 = 384$
7(Greek)	$96 \times 4 = 384$
8(Arabic)	$96 \times 4 = 384$
9(Hebrew)	$96 \times 4 = 384$
10(Cyrillic)	$96 \times 4 = 384$
11(Latin-5)	$96 \times 4 = 384$
12(GB2312-1980 中国語)	$94^2 \times 4 = 35344$
13(JISX0208-1983 日本語)	$94^2 \times 4 = 35344$
14(KSC5601-1987 韓国語)	$94^2 \times 4 = 35344$
合計	110616 バイト

となる。実際には 1 バイトあたり 94 または 96 個分のコードがすべて割り当てられているわけではないのでもう少し小さくできるだろう。

UTF 形式の場合、文字集合が連続的に割り当てられており、また日本語、中国語、韓国語の文字集合はそれぞれの国の標準規格 (それぞれ JISX0208, GB2312, KSC5601) とは全く異なったコードの割当がなされているので単純に文字が連続して割り当てられている部分をひとつのコードセットとする。例えば UTF-2 の場合は以下のようになる。

	第 1 バイト	第 2 バイト	第 3 バイト	サイズ (byte)
コードセット 0	0x00 – 0x7f			512
コードセット 1	0xc0 – 0xdf	0x80 – 0xbf		8192
コードセット 2	0xe0 – 0xe9	0x80 – 0xa7	0x80 – 0xbf	102400
コードセット 3	0xef – 0xef	0xa4 – 0xbf	0x80 – 0xbf	7168
			計	118272

後述するがさらにワイド→マルチバイト文字変換のテーブルも必要となるのでマルチバイト/ワイド文字相互変換のためにはこの倍以上のメモリを占領することになる。この程度であれば現在の標準的な計算機のメモリ資源ならそれほど厳しい要求ではないと思うがさらにメモリ消費を減らす方法として、

- setlocale 関数呼び出し時にすべてのテーブルをロードするのではなく、実際にマルチバイト→ワイド文字変換を行なう時に必要になったテーブルをダイナミックにロードする。
- テーブルを共有メモリ機構を用いてプロセス間で共有する。

といった方法も考えられる。

3.6.2 コードセットの判定

以上のような設計にしたことにより、マルチバイト文字からワイド文字に変換する前にマルチバイト文字がどのコードセットに属するかを判別して適切な変換テーブルを選択する必要がある。それで変換テーブルとは別に

- コードセット判定に必要な情報
- コードセットに関する情報 (1 文字のバイト数など)

が必要である。各ロケールの LC_CTYPE ファイルにはこのために必要な情報を記述する。それでマルチバイト→ワイド文字変換の大まかなながれは以下のようになる。

1. 1 マルチバイト文字のコードセットを判定する。
2. そのコードセットの 1 マルチバイト文字のバイト数を得る。
3. 1 マルチバイト文字分のバイト列を取り出す。
4. その文字の属するコードセットにあった変換テーブルの選択する。

5. ワイド文字に変換する。

コードセットの判定方法はステートレスエンコーディングの場合とステートフルエンコーディングの場合で大きく異なる。

ステートレスエンコーディングの場合

たいていのステートレスのコードは 1 バイト目の値によってコードセットが判定できる。例えば EUC であれば、

1 バイト目の値 (16 進)	コードセット	1 文字のバイト数
0x00 – 0x7f	0(ASCII)	1
0xa1 – 0xfe	1(JISX0208)	2
0x8e	2(JISX0201 カタカナ)	2
0x8f	3(JISX0212)	3

となる。しかし中にはコードセットが判明するまでに数バイト読まなければならないものも存在する。I-UTF では最大 3 バイト読まなければコードセットが判別できない。

1 バイト目	2 バイト目	3 バイト目	コードセット	1 文字のバイト数
0x00 – 0x9f			0	1
0xc0 – 0xdf			1	2
0xe0 – 0xef	0x21 – 0x28		2	2
	0x30 – 0x7e			
0xe0 – 0xef	0x80 – 0xbf	0x80 – 0xbf	3	3
0xe0 – 0xe7	0x80 – 0xbf	0x21 – 0x28	4	3
		0x30 – 0x7e		
0xe8 – 0xeb	0x80 – 0xbf	0x21 – 0x28	5	4
		0x30 – 0x7e		
0xec – 0xed	0x80 – 0xbf	0x21 – 0x28	6	5
		0x30 – 0x7e		
0xee	0x80 – 0xbf	0x21 – 0x28	7	6
		0x30 – 0x7e		
0xef	0x80 – 0xbf	0x21 – 0x28	8	7
		0x30 – 0x7e		
0xf0 – 0xff			9	2

このような文字コードによる違いに柔軟に対応できるコードセット判定ルーチンとロケールデータの記述方法を考える必要がある。

ステートフルなコードの場合

ステートフルなコードではエスケープシーケンスによってコードセットが切り替わるので、エスケープシーケンスが来たらそれに応じて 1 文字バイト数とマッピングテーブ

ルを切替えればよい。例えば CompoundText ではエスケープシーケンスとコードセットの関係は以下のようになる。

エスケープシーケンス	コードセット	バイト数
0x1b 0x28 0x42 (esc (B)	0(ASCII)	1
0x1b 0x29 0x49 (esc) I)	1(JISX0201-1976 カタカナ)	1
0x1b 0x28 0x4a (esc (J)	2(JISX0201-1976 ローマ字)	1
0x1b 0x29 0x41 (esc) A)	3(Latin-1)	1
0x1b 0x29 0x42 (esc) B)	4(Latin-2)	1
0x1b 0x29 0x43 (esc) C)	5(Latin-3)	1
0x1b 0x29 0x44 (esc) D)	6(Latin-4)	1
0x1b 0x29 0x46 (esc) F)	7(Greek)	1
0x1b 0x29 0x47 (esc) G)	8(Arabic)	1
0x1b 0x29 0x48 (esc) H)	9(Hebrew)	1
0x1b 0x29 0x4c (esc) L)	10(Cyrillic)	1
0x1b 0x29 0x4d (esc) M)	11(Latin-5)	1
0x1b 0x24 0x28 0x41 (esc $ (A)	12(GB2312-1980 中国語 Hanzi)	2
0x1b 0x24 0x28 0x42 (esc $ (B)	13(JISX0208-1983 日本語 Kanji)	2
0x1b 0x24 0x28 0x43 (esc $ (C)	14(KSC5601-1987 韓国語 Hangul/Hanja)	2

ステートフルのエンコーディングではエスケープシーケンスの後に続くマルチバイト文字の変換のために、設定されたコードセットと 1 文字のバイト数の情報を保存しておく必要がある。そのため MSE の仕様では 3.3.4で述べた通り、mbstate_t 型という新しい型を用意している。この型の変数にはどのような情報を入れるかは実装依存である。このライブラリでは mbstate_t は short 型と型定義し、単純に現在のコードセット番号を入れることにした。そしてコードセット番号と 1 文字のバイト数、そのコードセットに切り替わるエスケープシーケンスの対応リストは別に保持する。

3.7 ワイド→マルチバイト文字変換関数の設計

これに関係するのは以下の拡張マルチバイト/ワイド文字変換関数である。

```
wcrtomb    wcsrtombs  wctob
```

前述のようにワイド文字コードは ICODE なので EUC や JIS とはコードに対する文字集合の割り当て方が大きく異なる。そのためワイド文字からマルチバイト文字への変換アルゴリズムもマッピングテーブルを用いて行なう必要がある。ICODE での文字集合の割り当ては図 3.6のようになっている。

この影響でマルチバイト文字コードが JIS コードや日本語 EUC のロケールの場合、JISX0208 の文字集合に属する文字がワイド文字ではさまざまなところにマッピングされている。また漢字が割り当てられている部分では JISX0208 と JISX0212 に含まれる

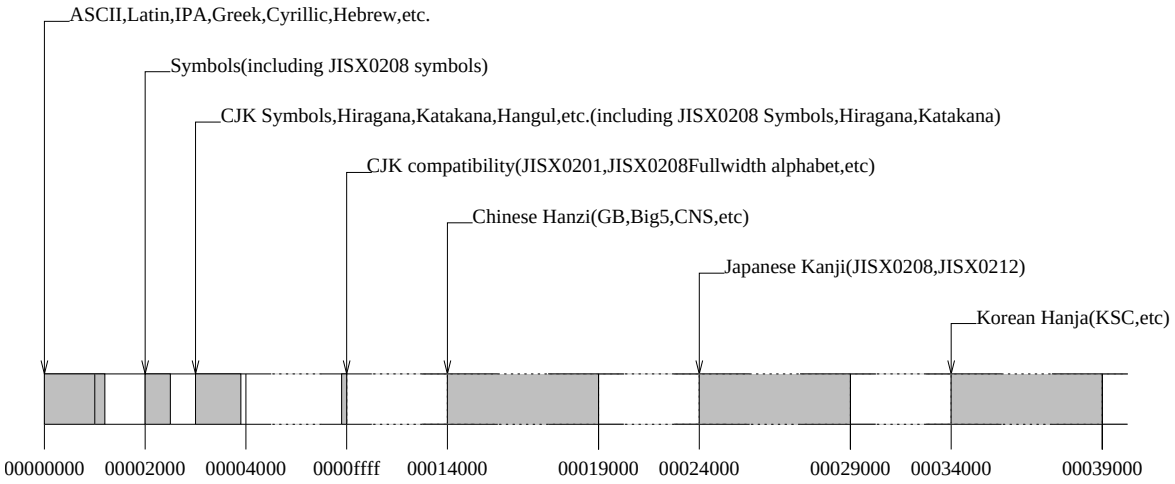


図 3.6: ICODE のコード割り当て

文字が混在している。このことはマルチバイト文字がステートレスの文字コードの場合は特に問題にならないが、ステートフルの文字コードの場合 (この場合は JIS コード) は変換されてできたマルチバイト文字がどの文字集合に属するかを判定し、それが直前に変換した文字の属する文字集合と異なる場合はエスケープシーケンスを前につけてやらなければならない。それでワイド→マルチバイト文字変換マッピングテーブルの各エントリには

- 変換されてできるマルチバイト文字
- そのマルチバイト文字のコードセット

の情報が格納されるようにする。そしてコードセットとバイト数、エスケープシーケンスの組も別に格納しておく。

またワイド文字コード空間のすべてのコードに文字が割り当てられているわけではないし、ワイド文字にはロケールによってはマルチバイト文字に変換不可能な文字も含まれる。たとえば JIS コードのロケールではヘブライ文字やアラビア文字は存在しない。それでそのロケールのマルチバイト文字コードに対応するワイド文字コードの連続している部分をグループとして抽出する。JIS、日本語 EUC、シフト JIS のロケールの場合は以下のようになる。

	ワイド文字コードの範囲	対応するコードセット
グループ 0	0x00000000–0x000000f7	ASCII & latin-1
グループ 1	0x00024e00–0x00029fa5	漢字 (JISX0208 & JISX0212)
グループ 2	0x00000401–0x0000045f	キリル文字 (JISX0208)
グループ 3	0x00000391–0x000003c9	ギリシャ文字 (JISX0208)
グループ 4	0x00002010–0x0000203e	記号 (JISX0208)
グループ 5	0x00002103–0x00002103	記号 (JISX0208)
グループ 6	0x00002190–0x00002193	記号 (JISX0208)
グループ 7	0x000021d2–0x00002312	記号 (JISX0208)
グループ 8	0x00002500–0x0000274d	記号 (JISX0208)
グループ 9	0x00003000–0x000030fe	記号、ひらがな、カタカナ (JISX0208)
グループ 10	0x0000ff01–0x0000ff9f	半角カナ (JISX0201)、全角アルファベット (JISX0208)
グループ 11	0x000002b9–0x000002ba	記号 (JISX0208)

ワイド→マルチバイト文字変換テーブルはこのグループごとに用意される。これによってテーブルに必要なメモリ領域を節約する。しかし 1 エントリの大きさがマルチバイト→ワイド文字変換テーブルより大きくなる可能性がある (マルチバイト文字のバイト列とそのコードセット、バイト数が必要である) こと、グループ化しても各グループ内で文字の割り当てられていないコードが多いのでマルチバイト→ワイド文字変換テーブルより大きくなると予想される。

3.8 ワイド文字入出力関数の設計

これに関係する関数は以下の通りである。

```
fgetwc    fgetws    getwc     getwchar  fwscanf   wscanf    swscanf
fputwc    fputws    putwc     putwchar  fwprintf  wprintf   swprintf
vfwprintf vwprintf  vswprintf ungetwc   fwide
```

ワイド文字入力関数はバイトストリームからバイト列を読み込んで内部でワイド文字に変換してワイド文字 (列) に格納し、ワイド文字出力関数はワイド文字 (列) をマルチバイトバイト文字列に変換してバイトストリームに出力する。これらの関数は内部で拡張マルチバイト/ワイド文字変換関数を呼び出すことによってこれを行なうことにする。

ステートフルエンコーディングのストリームに入出力する場合はストリームごとにステートを保存しなくてはならない。これも拡張マルチバイト/ワイド文字変換関数の場合のようにステートを保存する変数を明示するのではなく、関数内部で行なう。こ

うすれば従来の標準入出力ライブラリ関数とほとんど同一のインタフェースを実現できる。これは FILE 構造体の中にシフトステートを示す mbstate_t 型のメンバを付け加えればよさそうだがシステム標準の標準入出力関数への影響を考えるとこれはあまり良い方法ではない。さらに MSE ではストリームごとに 『オリエンテーション」を保存することを規定している。つまりいったんあるストリームにワイド入出力関数で入出力したらそのストリームは 『ワイドオリエンテーション」となる。同様にいったん標準入出力関数で入出力したら 『バイトオリエンテーション」となる。ワイドオリエンテーションとなったストリームに標準入出力関数を適用することはできず、バイトオリエンテーションのストリームにワイド入出力関数を適用することはできない。このオリエンテーションの確認、変更のために fwide 関数が用意されている。それで各ファイルポインタと現在のステート (つまりコードセット) とオリエンテーションの対応テーブルを用意し、各ワイド文字入出力関数が参照できるようにすることにした。

3.9 ワイド文字分類関数の設計

これに関係する関数は以下の通りである。

```
iswalnum  iswalpha  iswdigit  iswgraph   iswlower  iswprint
iswpunct  iswspace  iswupper  ixwxdigit  wctype    iswctype
```

これもロケールによって動作を変える。 MSE では標準ライブラリの isalpha,islower など文字分類関数のワイド文字対応版 (iswalpha, iswlower など) のほかに、iswctype 関数と wctype 関数、そして wctype_t という型が追加された。それぞれ以下のような機能を持つ。

- **wctype_t**... ロケール依存の文字分類情報を保持するスカラー型
- **wctype_t wctype(const char *property);**... 文字分類名 property を与えると property に関する分類情報を返す
- **int iswctype(wint_t wc, wctype_t desc);**... desc に指定された分類情報をもとにワイド文字 wc がその分類に当てはまるかどうかチェック

その他のワイド文字分類関数は wctype 関数と iswctype 関数を用いて記述することができる。例えば iswalnum(wc) は iswctype(wc, wctype("alnum")) と等価である。ロケール定義ファイルに分類名とその分類情報を記述しておけば、iswctype と wctype を用いて新たな文字分類を作成することも可能である。

まず wctype_t 型は具体的にはどのような情報を保持するかを決めなくてはならない。多くの場合、ある分類に当てはまる文字はコード的に連続した領域を占めている。それで

分類に当てはまるコードの下限 1	分類に当てはまるコードの上限 1
分類に当てはまるコードの下限 2	分類に当てはまるコードの上限 2
分類に当てはまるコードの下限 3	分類に当てはまるコードの上限 3
⋮	⋮
分類に当てはまるコードの下限 n	分類に当てはまるコードの上限 n

という表へのポインタとした。wctype 関数は現在のロケールで、指定された分類名に対応する wctype_t の値を返す。iswctype 関数は与えられたワイド文字コードがこの表で指定された範囲内にあるかどうかをチェックする。

3.10 ワイド文字変換関数の設計

これに関係するのは以下の関数である。

```
towlower towupper wtrans   towtrans
```

MSE では標準ライブラリの tolower,toupper のワイド文字対応版である tolower と toupper が用意されている。これらも文字分類関数はロケールによって動作を変える。また、ワイド文字分類関数と同様に機能拡張用として towctrans 関数と wctrans 関数、そして wctrans_t という型が追加された。それぞれ以下のような機能を持つ。

- **wctrans_t**... ロケール依存の文字変換情報を保持するスカラー型
- **wctrans_t wctrans(const char *property);**... 文字変換名 property を与えると property に関する変換情報を返す
- **wint_t towctrans(wint_t wc, wctrans_t desc);**... desc に指定された変換情報をもとにワイド文字 wc を変換する

towlower(wc),towupper(wc) は wctrans 関数と iswctrans 関数を用いてそれぞれ、

towctrans(wc, wctrans("tolower"));
towctrans(wc, wctrans("toupper"));

と記述することができる。

wctrans_t 型については wctype_t 型にその範囲のコードがどう変換されるかを示す項目を付加した形になる。それで wctrans_t 型は

コードの下限 1	コードの上限 1	変換先 1
コードの下限 2	コードの上限 2	変換先 2
コードの下限 3	コードの上限 3	変換先 3
⋮	⋮	⋮
コードの下限 n	コードの上限 n	変換先 n

という表へのポインタとした。wctrans 関数は現在のロケールで、指定された変換名に対応する wctrans_t の値を返す。iswctrans 関数は与えられたワイド文字コード wc がこの表の i 番目の範囲内にあった場合、

変換先 $i + wc -$ コードの下限 i

という計算によって変換を行なう。

3.11 ワイド文字列関数群の設計

ここに分類されるのは以下の関数である。

wcstod	wcsncpy	wcscoll	wcscspn	wcsstr	wmemcmp
wcstol	wcscat	wcsncmp	wcspbrk	wcstok	wmemcpy
wcstoul	wcsncat	wcsxfrm	wcsrchr	wcslen	wmemmove
wcscpy	wcscmp	wcschr	wcsspn	wmemchr	wmemset

これらはワイド文字に対する操作だけを考えればよく、ロケールの影響は受けない (例外は wcscoll、wcsxfrm、wcsftime であるがこれらはロケールカテゴリが LC_CTYPE ではない– wcscoll, wcsxfrm は LC_COLLATE、 wcsftime は LC_TIME)。これらは標準ライブラリの対応する関数が char 型の変数に対する処理だったところを wchar_t 型に対する処理に変更すればよいだけである。

第 4 章

ライブラリの実装

以上の設計を元に各ライブラリ関数の実装を行なう。

4.1 内部でのロケール情報の保持

マルチバイト↔ワイド文字変換関数、ワイド文字分類関数、ワイド文字変換関数が必要とするロケール依存情報を保持する構造体_lc_ctype を定義した。この構造体は 4.1 のようになっている。各関数はこの構造体へのグローバルなポインタ*_lcctypeinfo によってこの内容を参照することができる。 setlocale 関数はファイル (指定されたロケールの LC_CTYPE ファイル) からこの構造体にデータをロードする。

この構造体からそれぞれの関数に必要な情報を保持した構造体へのポインタがのびているがそれらについてはそれぞれの関数の実装の説明のときにに詳述する。

4.2 拡張マルチバイト→ワイド文字変換関数の実装

3.6章で述べたとおり、マルチバイト→ワイド文字変換関数は

1. コードセット数判定ルーチン
2. マルチバイト→ワイド文字変換ルーチン

から成る。バイト数判定ルーチンはステートレスエンコーディングかステートフルエンコーディングかで大きく異なる。それで mbrtowc 関数の内部関数の呼びだし関係は図 4.2のようになった。mbrlen 関数もこれと同様となる。 mbsrtowcs 関数もこれら内部ルーチンを繰り返し呼ぶ以外は同様の構造をしている。

4.2.1 マルチバイト→ワイド文字変換アルゴリズム

_mbrtowc_trans 内部関数についてであるが、この部分はステートレスであってもステートフルであっても共通である。マッピングテーブルの実現方法とコードセットご

```
typedef struct __lc_ctype {
    char        *_localename;   /* ロケール名 */
    short        _mb_cur_max;   /* 1 マルチバイト文字の最大バイト数 */
    short        _encoding;     /* ステートフル:1 ステートレス:0 */
    char         _esc;          /* エスケープ文字（ステートフルの場合）*/
    short        _ncs;          /* コードセットの数 */
    short        _nwcmbtab;     /* ワイド文字グループ数 */
    short        _ntype;        /* ワイド文字分類の数 */
    short        _ntrans;       /* ワイド文字変換の数 */
    _mbwctab_sf *_mbwctab_sf;   /* マルチバイト→ワイド変換テーブルへのポインタ（ステート
フル用）*/
    _mbwctab_sl *_mbwctab_sl;   /* マルチバイト→ワイド変換テーブルへのポインタ（ステート
レス用）*/
    _csinfo     *_csinfo;       /* コードセット情報テーブルへのポインタ */
    _wcmbtab    *_wcmbtab;      /* ワイド→マルチバイト変換テーブルへのポインタ */
    _wctypetab  *_typeinfo;     /* ワイド文字分類テーブルへのポインタ */
    _wctranstab *_transinfo;    /* ワイド文字変換テーブルへのポインタ */
    size_t     (*_mbrtowc)();   /* マルチバイト→ワイド文字変換関数へのポインタ */
    size_t     (*_wcrtomb)();   /* ワイド→マルチバイト文字変換関数へのポインタ */
} _lc_ctype;
```

図 4.1: _lc_ctype 構造体

とのマッピングテーブルの管理方法を考慮する必要がある。それで図 4.3のようなデータ構造を用いることにした。

各マッピングテーブルは

```
typedef union __mbwc {
    wchar_t         _wc;    /* 対応するワイド文字 */
    union __mbwc  *_next; /* 次のバイトのマッピングテーブルへのポイン
タ */
} _mbwc;
```

という共用体が (そのバイト範囲の最大値 - 最小値 + 1) 個並んだ配列である。共用体にしたのはテーブルのメモリ消費量をできるだけ減らすためである。またコードセットテーブルは以下のような構造体がコードセットの数だけ並んだ配列である。この構造体は_lcctypeinfo->_csinfo から参照できる。

```
struct __csinfo {
    short  _mblen; /* 1 マルチバイト文字のバイト数 */
    _mbwc  *_trans; /* 第 1 バイトマッピングテーブルへのポインタ */
    struct {  /* 各バイト範囲 */
        unsigned char _begin;  /* 最小値 */
        unsigned char _end;    /* 最大値 */
    } _range[_MAX_MB_LEN];
} _csinfo;
```

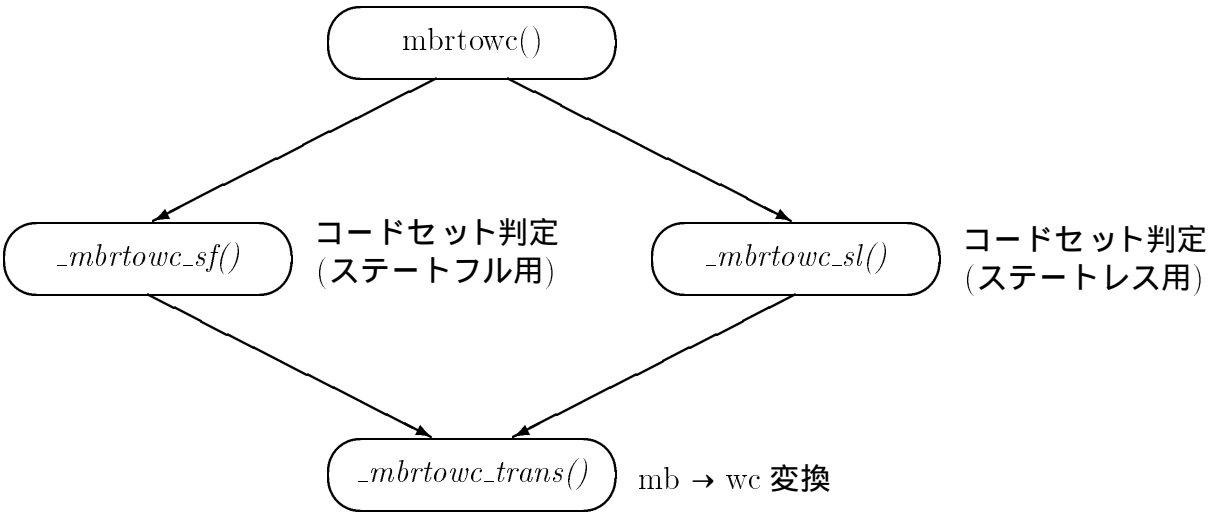


図 4.2: mbrtowc 関数の内部関数

変換にはまずそのマルチバイト文字の属するコードセットを判別し (このアルゴリズムについては後述する)、1 バイトずつそのコードセットの各バイトの範囲内にあるかどうかチェックしながらテーブルをたどる。そのコードセットの 1 文字のバイト数分テーブルをたどると対応するワイド文字に行きつくという仕組みである。もう少し C らしく基本的なアルゴリズムを記述すると以下のようになる。

```
/*
  _mbrtowc_trans
    *mb:マルチバイト文字を指す char 型ポインタ
     wc:対応するワイド文字
     table:マッピングテーブルへのポインタ
*/
_mbrtowc_trans(*wc, *mb, コードセットテーブル)
{
    for(i = 1; i < そのコードセットのバイト数; i++, mb++){
        if(*mb の値が第 i バイト範囲の最大値と最小値の間にある)
            table = table[*mb - 第 i バイト範囲の最小値]._next;
        else
            return(エンコーディングエラー);
        if(table == -1)
            return(エンコーディングエラー);
    }
    *wc = table[*mb - 第 i バイト範囲の最小値]._wc;
}
```

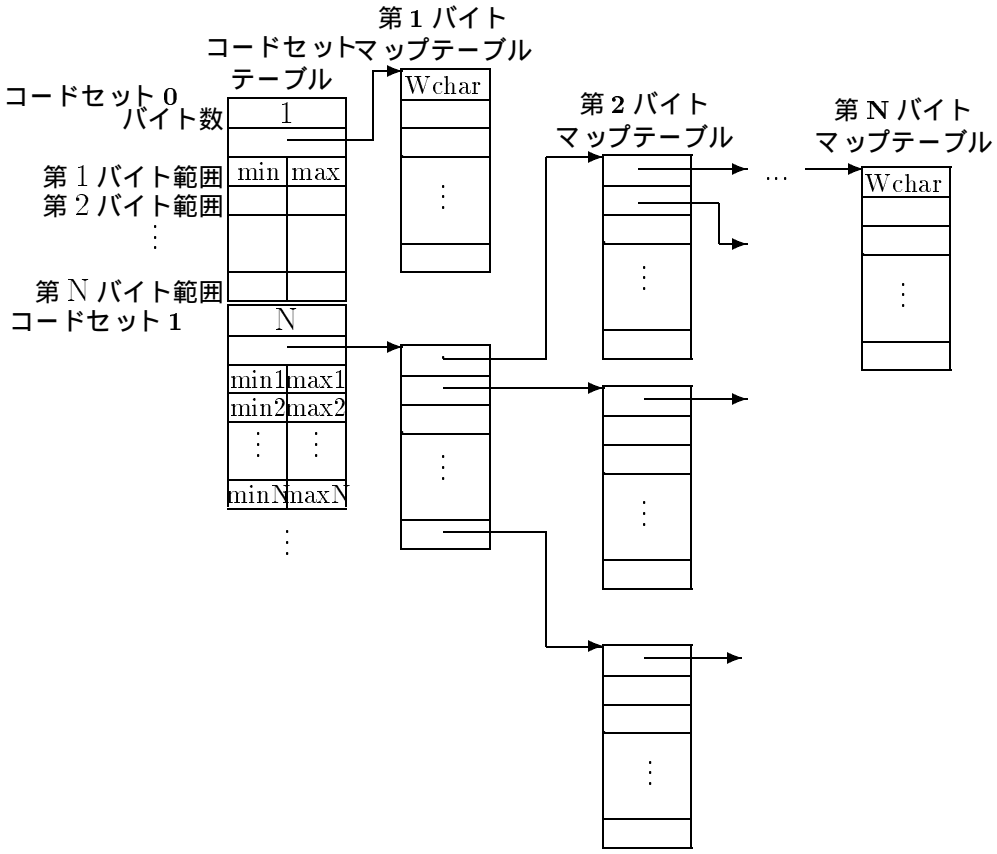


図 4.3: マルチバイト→ワイド文字変換用データ構造

4.2.2 コードセット判別アルゴリズム

マルチバイト→ワイド文字変換ルーチンにはいる前にコードセットの判別が必要であることは何度か指摘した。これを行なう内部ルーチン _mbrtowc_sl, _mbrtowc_sf についてさらに説明を加える。

ステートレスエンコーディングの場合

コードセットが確定するまでに読まなければならないバイト数が何バイトでもよいような柔軟なアルゴリズムとデータ構造を考える必要がある。それで図 4.4のようなデータ構造を考えた。

これも一種のマッピングテーブルで各バイトの値をそのまま添字にあたえる。各バイトごとのテーブルは

```
typedef struct __mbwctab_sl {
    struct __mbwctab_sl  *_next;     /* 次のバイトのテーブル */
    _csinfo              *_csinfotab; /* コードセットテーブル */
} _mbwctab_sl;
```

という構造体を 256 個持つ配列となっている。このテーブルは_lcctypeinfo->_mbwctab_sl

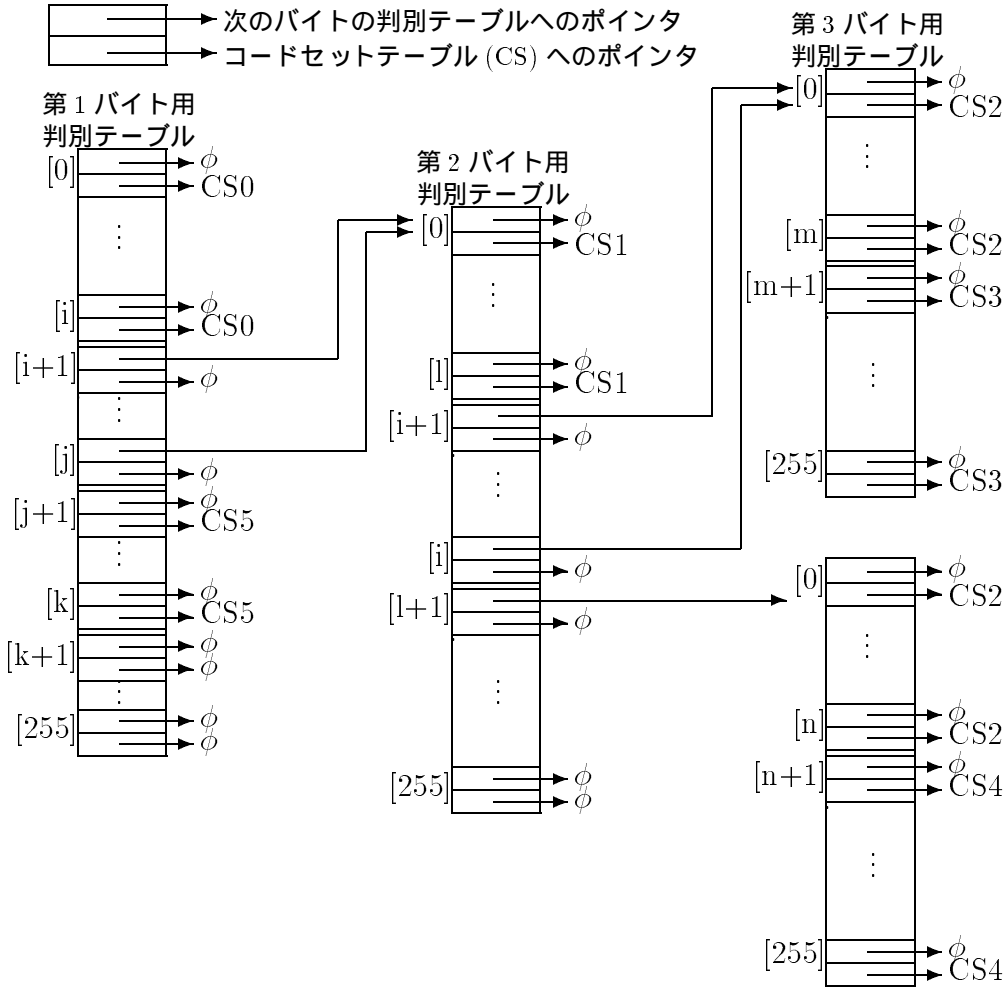


図 4.4: コードセット判別テーブル (ステートレス)

から参照できる。これを用いたコードセット判別アルゴリズムは以下のようなものになる。

```
/*
    *p, *mb: マルチバイト文字へのポインタ
    *wc: ワイド文字へのポインタ
    *mbwctab: コードセット判別テーブルへのポインタ
*/
_mbtowc_sl(*wc, *mb)
{
    p = mb;
    *mbwctab = _lcctypeinfo->_mbwctab_sl[*p];
    while(mbwctab->_next != NULL) /* 次のバイト用テーブルがある間 */
        mbwctab = mbwctab->_next[*++p]; /* 次のバイトのテーブルエントリへ */
```

```
    if(mbwctab->_csinfotab == NULL) /* コードセットテーブルがない */
        return(エラー);
    _mbwctab_trans(wc, mb, mbwctab->_csinfotab);
}
```

このようなデータ構造にするとメモリ消費が気になるが、この構造体は一つで 8 バイトであるからひとつのテーブルは $8 \times 156 = 2028$ バイト消費する。ほとんどのコードは第 1 バイトでコードセットが判別でき、コードセット判別テーブルは 1 つだけですむ。また I-UTF の場合でも 11 個のテーブルがあればよいので 22528 バイトとなる。

```
for(i = 0; i < 現在のロケールのコードセット数; i++){
    if(コードセット i の最小値 <= *mb && *mb < コードセット i の最大値)
        _mbrtowc_trans(...);
}
```

というアルゴリズムも考えられるが、判別スピードを重視してマッピングテーブルを用いる方法を採用した。

ステートフルエンコーディングの場合

ステートフルエンコーディングのコードセット判別情報の格納には以下のような構造体の配列を使用することにした。

```
typedef struct __mbwctab_sf {
    short    _cs;                   /* コードセット番号 */
    char     _seqstr[_MAX_SEQ_LEN]; /* エスケープシーケンス */
    short    _seqlen;               /* エスケープシーケンスのバイト数 */
    _csinfo *_csinfotab;            /* コードセットテーブルへのポイン
タ */
} _mbwctab_sf;
```

例えば ja_JP.jis7(日本語 JIS コード) のロケールの場合、図 4.5のようになる。

このテーブルへのポインタが_lcctypeinfo->_mbwctab_sf である。それでステートフルの場合のコードセットの判別は以下のようなアルゴリズムになる。

```
/*
    *ps: ステートを保存する mbstate_t 型の変数へのポインタ
*/
_mbrtowc_sf(*wc, *mb, *ps)
{
    char *p = mb;
    _mbwctab_sf *mbwctab = _lcctypeinfo->_mbwctab_sf;
```

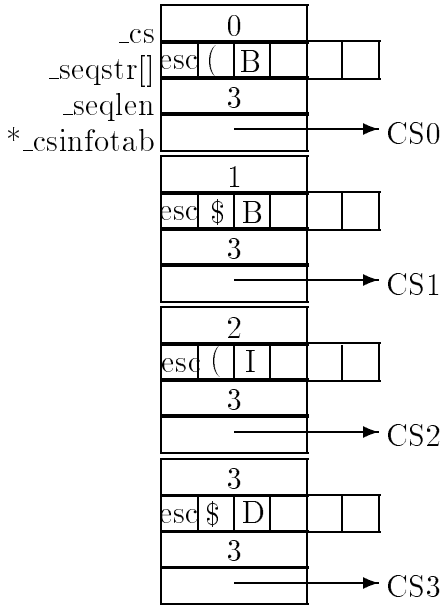


図 4.5: ステートフル用コードセット判別テーブル

```
    if(*p == _lcctypeinfo->_esc){ /* mb が指しているのがエスケープ文字
か */
        /* コードセットの数だけループ */
        for(cs = 0; cs < _lcctypeinfo->_ncs; cs++, mbwctab++){
            if(strncmp(p, mbwctab->_seqstr, mbwctab->_seqlen) == 0){
                /* エスケープシーケンスとマッチした */
                *ps = mbwctab->_cs; /* 新しいステートを保存 */
                p += mbwctab->_seqlen; /* エスケープシーケンスを読み
飛ばす */
                break;
            }
        }
    }else /* 現在のステート（コードセット）を選択 */
            mbwctab = _lcctypeinfo->_mbwctab_sf + *ps;

    _mbtowc_trans(wc, mb, mbwctab->_csinfotab);
}
```

コードセット判別ルーチンの切替え

エンコーディング方法によってバイト数判別ルーチンの切替えをしなければならないが、それを mbrtowc 関数で

```
if(_lcctypeinfo->_encoding == 0) /* ステートレスの場合 */
    _mbrtowc_sl();
else /* ステートフルの場合 */
    _mbrtowc_sf();
```

などとやっているとオーバヘッドが大きいので、setlocale 関数でロケールデータをロードした時に

```
if(_lcctypeinfo->_encoding == 0) /* ステートレスの場合 */
    _lcctypeinfo->_mbrtowc = _mbrtowc_sl;
else /* ステートフルの場合 */
    _lcctypeinfo->_mbrtowc = _mbrtowc_sf;
```

のようにして共有の構造体にエンコーディングに適したルーチンを登録し、mbrtowc 関数では

```
_lcctypeinfo->_mbrtowc();
```

と呼び出すことができるようにした。

4.3 拡張ワイド→マルチバイト文字変換関数の実装

これもマルチバイト→ワイド文字変換関数と同様にステートフルエンコーディングの場合とステートレスエンコーディングの場合で動作が異なる。それで拡張ワイド→マルチバイト文字変換関数は内部関数_wcrtomb_sl()(ステートレス用) もしくは_wcrtomb_sf()(ステートフル用) を呼び出す。これも setlocale 関数がロケールデータをロードした時にいずれかが_lcctypeinfo->_wcrtomb にセットされる。

おおまかな処理のながれは以下の通りである。

1. そのワイド文字がどのワイド文字グループに属するか判別する
2. ワイド文字をそのグループの変換テーブルにあてはめ、マルチバイト文字とそのコードセットを得る
3. ステートフルエンコーディングの場合、そのコードセットが直前のワイド文字のコードセットと異なっていたらそのコードセットに切替えるエスケープシーケンスをまずバイト列に書き込む
4. バイト列にマルチバイト文字を書き込む

これを実現するためワイド→マルチバイト文字変換情報を保持する_lcctypeinfo->_wcmbtab は図 4.6のようなデータ構造へのポインタとなっている。

変換の時に必要になるマルチバイト文字のバイト数とステートフルエンコーディングの場合のエスケープシーケンスは、マルチバイト→ワイド文字変換の時に用いたテーブル_lcctypeinfo->_csinfo、_lcctypeinfo->_mbwctab_sl から得ることができる。

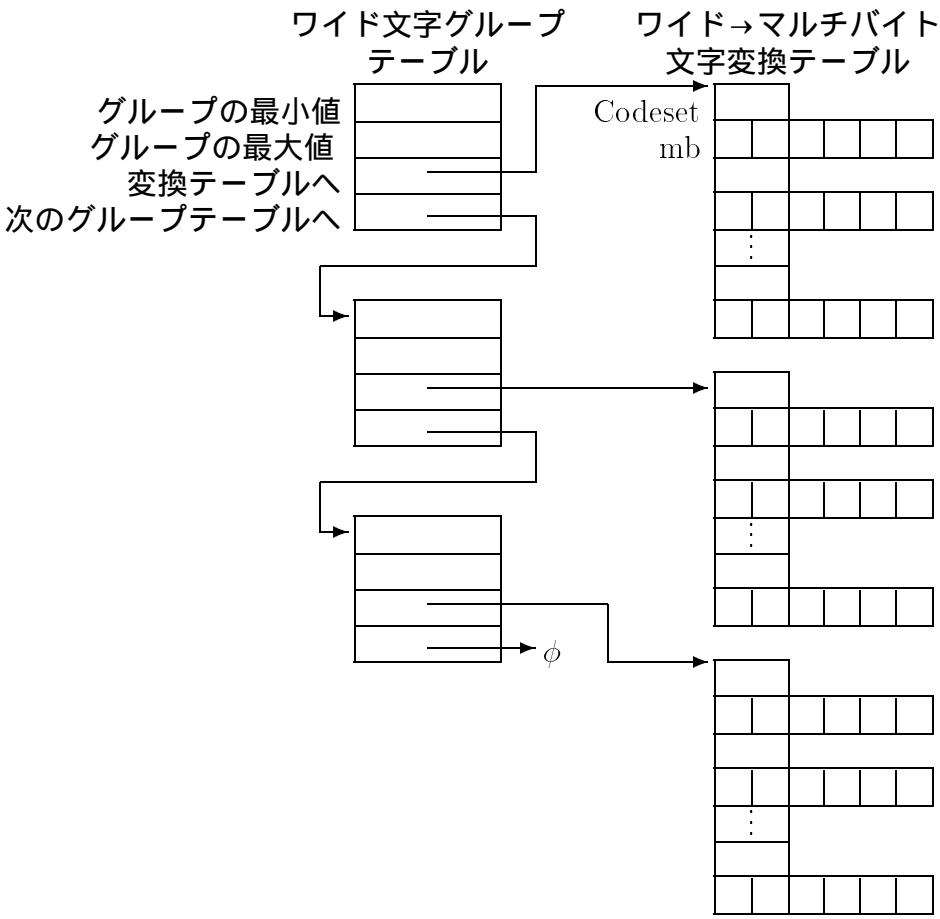


図 4.6: ワイド→マルチバイト文字変換テーブルの構造

ワイド文字グループの検索にかかる時間を短縮するためここでは同じグループの文字は続けて出る可能性が多いと考えられるので、ワイド文字グループテーブルの先頭 (つまり_lcctypeinfo->_wcmbtab が直接指しているエントリ) 以外のエントリに現在注目しているワイド文字が属していた場合、そのエントリがワイド文字グループテーブルの先頭にくるようにポインタのつけ替えをするようにした。それまで先頭だったエントリは 2 番目に下がる。これで出現頻度の高いエントリが早く検索できると考えられる。

4.4 ワイド文字 I/O 関数群の実装

これらの内部構成は大きく

- バイト入力/出力ルーチン
- マルチバイト/ワイド文字変換ルーチン

に分けることができる。

バイト入力/出力ルーチンには最初は標準入出力ライブラリ関数が内部で用いている低レベルルーチン (BSD/386 では__swbuf など) を用いることを考えていたがシステムによってかなり相違があり、移植性が悪くなるので標準入出力ライブラリ関数を用いることにした。そしてマルチバイト/ワイド文字変換ルーチンには拡張マルチバイト/ワイド文字変換関数を用いる。

ストリームごとのステートとオリエンテーションの保存のために

```
typedef struct __stream_info {
    FILE         *_stream; /* ファイルポインタ */
    mbstate_t     _state;  /* ステート */
    short         _orient; /* オリエンテーション
                                  1:ワイドオリエンテーション
                                 -1:バイトオリエンテーション
                                  0:オリエンテーションなし */
} _stream_info;
```

という構造体の配列_stream_infotab を用意した。これはそのシステムで一つのプロセスがオープンできる最大ファイル数個分の要素を持つ。オープンできる最大ファイル数はシステムによって異なるが何らかのインクルードファイルに定義されている。たとえば BSD/386 の場合は/usr/include/sys/syslimits.h に OPEN_MAX として 64 が定義されている。そしてあるストリーム fp に関する情報は_stream_infotab[fileno(fp)] で参照することができる。fileno 関数はそのストリームに対応するファイルディスクリプタ番号を返すライブラリ関数である。

それでワイド入出力関数の処理のながれは以下のようになる。

- ワイド入力関数 (fgetwc)

```
fgetwc(FILE *stream)
{
    mbstate_t *ps;
    wchar_t wc; /* ワイド文字 */
    char mb[MB_LEN_MAX]; /* マルチバイト文字バッファ
                            (MB_LEN_MAX:1 マルチバイト文字の最大バイト数) */
    size_t len; /* 読み込んだバイト数 */
    size_t bytelen; /* 読み込んだ 1 マルチバイト文字のバイト数 */

    /* オリエンテーションのチェック */
    if(_stream_infotab[fileno(stream)]._orient == 0)
        /* オリエンテーションなしならワイドオリエンテーションに */
        _stream_infotab[fileno(stream)]._orient == 1;
    else if(_stream_infotab[fileno(stream)]._orient == -1)
```

```
        /* バイトオリエンテーションならエラー */
        return(ERROR);
    ps = &_stream_infotab[fileno(stream)]._state; /* ストリーム
のステートの取得 */
    len = fread(mb, 1, MB_CUR_MAX, stream); /* ストリームからマ
ルチバイト文字を読む */
    bytelen = mbrtowc(&wc, mb, len, ps); /* ワイド文字へ変換 */

    while(len-- > bytelen)      /* 読み過ぎた分をストリームに戻す */
        ungetc(*--p, stream);
    return(wc);
}
```

- ワイド出力関数 (fputwc)

```
fpetwc(wchar_t wc, FILE *stream)
{
    mbstate_t *ps;
    char mb[MB_LEN_MAX]; /* マルチバイト文字バッファ
                            (MB_LEN_MAX:1 マルチバイト文字の最大バ
イト数) */
    size_t bytelen; /* 1 マルチバイト文字のバイト数 */

    オリエンテーションのチェック; /* fgetwc と同様 */
    ps = &_stream_infotab[fileno(stream)]._state; /* ストリーム
のステートの取得 */
    bytelen = wcrtomb(mb, wc, ps); /* マルチバイト文字に変換 */
    fwrite(mb, 1, bytelen, stream); /* ストリームへ出力 */
}
```

4.5 ワイド文字分類関数群の実装

4.5.1 wctype 関数の実装

wctype 関数は分類属性名を引数にとり、その分類属性についての分類テーブル (wctype_t 型) を返す。それで分類属性名と分類テーブルの組を保持するテーブルを用意し、_lcctypeinfo->_typeinf からこれを参照できるようにした。このテーブルは図 4.7のような構造体の配列になっている。

基本的なアルゴリズムは以下のようになる。

```
wctype_t wctype(const char *property)
```

```
typedef struct __wctypetab {
    char       _prop[15];          /* 分類属性名 */
    wctype_t   _typetab;           /* 分類テーブル */
} _wctypetab;
```

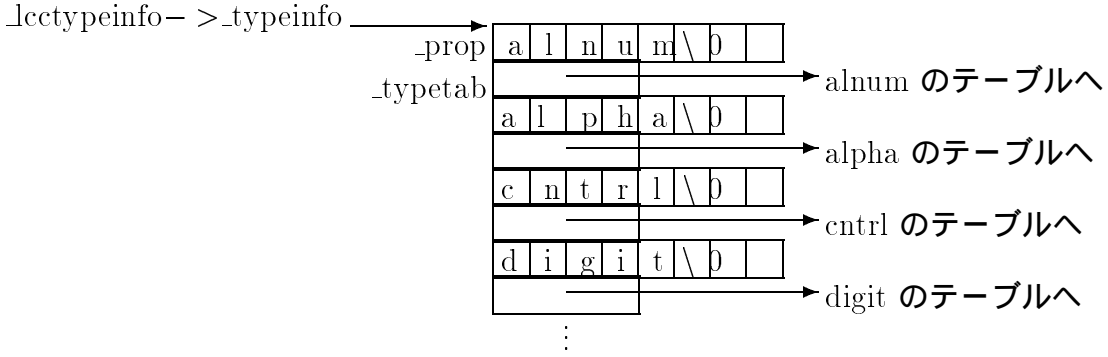


図 4.7: 分類属性テーブルの構造

```
{
    _wctypetab *ptr;     /* 分類属性テーブルへのポインタ */
    int i;

    ptr = _lcctypeinfo->_typeinfo;
    for(i = 0; i < _lcctypeinfo->_ntype; i++, ptr++){
        if(strcmp(property, ptr->_prop) == 0) /* 属性名の検索 */
            return(ptr->_typetab); /* テーブルを返す */
    }
    return(0); /* その属性名が見つからなかった */
}
```

説明を簡単にするために属性名の検索は線形検索法で記述したが実際にはこのような線形検索では時間がかかり過ぎると考え、二分検索のアルゴリズムで実装した。

4.5.2 iswctype 関数の実装

分類テーブルを保持する wctype_t 型を以下のように定義した。

```
typedef struct __wctype_t {
    wchar_t  _low;              /* 範囲の最小値 */
    wchar_t  _high;             /* 範囲の最大値 */
} * wctype_t;
```

ここで問題になるのは分類テーブルの大きさがわからないことである。分類テーブルの大きさは分類属性名と組で保持すればよさそうだが、iswctype 関数には wctype_t 型

の変数 (つまり分類テーブルへのポインタ) しか渡らない。それであまりきれいな方法ではないが、分類テーブルの最初のエントリにテーブルの大きさをいれることにした。

大まかな処理のながれは以下のようになる。

```
int iswctype(wint_t wc, wctype_t desc)
/* wc: ワイド文字
   desc: 分類テーブル */
{
    int i;
    wctype_t ptr = desc + 1;

    /* 最初のエントリ (desc->_low) はエントリ数 */
    for(i = 0; i < desc->_low; i++, ptr++)
        if(desc->_low <= wc && desc->_high)
            return(1); /* 真 */
    return(0); /* 偽 */
}
```

wctype 関数と同様、実際には二分検索を用いている。

4.6 ワイド文字変換関数群の実装

ワイド文字変換用のデータ構造、アルゴリズムはワイド文字分類用のそれとほとんど同一で、

```
typedef struct __wctranstab {
    char       _prop[15];       /* 変換属性名 */
    wctrans_t  _transtab;       /* 変換テーブル */
} _wctranstab;
```

という構造体の配列となっている。これは_lcctypeinfo->_transinfo から参照することができる。

wctrans_t 型は以下のような構造体へのポインタとした。

```
typedef  struct __wctrans_t {
    wchar_t  _low;              /* 範囲の最小値 */
    wchar_t  _high;             /* 範囲の最大値 */
    wchar_t  _to;               /* 変換先の最小値 */
} * wctrans_t;
```

wctrans 関数は wctype 関数と同様のアルゴリズムで引数として与えられた変換属性名にあった変換テーブルを返す。towctrans 関数も iswctype 関数とほぼ同様のアルゴリズムを採用した。処理のながれは以下のようである。

```
wint_t towctrans(wint_t wc, wctrans_t desc)
/* wc: ワイド文字
   desc: 変換テーブル */
{
    int i;
    wctrans_t ptr = desc + 1;

    /* 最初のエントリ (desc->_low) はエントリ数 */
    for(i = 0; i < desc->_low; i++, ptr++)
        if(desc->_low <= wc && desc->_high)
            return(wc - ptr->_low + ptr->_to);
    return(wc);
}
```

4.7 ワイド文字列関数群の実装

ほとんどの場合、標準ライブラリ関数の “str”、“mem” で始まる名前の関数をそれぞれ “wcs”、“wmem” で始まる名前に変更し、char 型と宣言されていたところを wchar_t 型に変更して、インクルードファイルを string.h から wchar.h に変更するだけですむ。

例えば strlen 関数は

```
#include <string.h>
size_t
strlen(register const char *str)
{
    register const char *s;

    for(s = str; *s; s++)
        ;
    return(s - str);
}
```

のようになっているが、これのワイド文字版である wcslen は

```
#include <wchar.h>
size_t
strlen(register const wchar_t *str)
{
    register const wchar_t *s;

    for(s = str; *s; s++)
```

```
        ;
    return(s - str);
}
```

とすればよい。最初は Berkley network release 2 の標準ライブラリをもとに作り替える予定であったが、それほど難しくないのでスクラッチから書いた。

4.8 ロケール定義ファイル

ロケール定義ファイルはテキスト形式でできるだけ記述しやすいものとした。以下のような形式で記述する。基本的には

セクション名{
　データ並び 1
　データ並び 2
　　...
　データ並び n
}

という形式で記述し、データ並びには

- エントリ名... 文字列+数字
- 10 進定数... n (n は任意の桁の 10 進数)
- バイト定数... '\xHH' (HH は 2 桁までの 16 進数)
- ワイド文字定数... L'\xHHHHHHHH' (HHHHHHHH は 8 桁までの 16 進数)
- バイトストリング... "\xHH\xHH\xHH..." (任意の長さの\xHH の並び)
- コメント... # (#から行の終わりまでがコメント)

をもちいることができる。

```
#  (#以降改行まではコメント)
generalinfo { #各種定数設定
#定数名          値
ENCOIDNG         STATEFUL または STATELESS     #エンコーディングの指定
ESC              '\xHH'  #ステートフルの場合のエスケープ文字の指定 (HH は 2 桁の 16 進
数)
MB_CUR_MAX       n #このロケールでの 1 マルチバイト文字の最大バイト数 (n は任意の 10 進
数)
N_CS             n  #コードセットの数
N_WCGROUP        n  #ワイド文字グループの数
N_TYPE           n  #ワイド文字分類の数
N_TRANS          n  #ワイド文字変換の数
}
```

```
csjudgeinfo{ #コードセット判定テーブル（ステートレス用）
#1st range : 2nd range : 3rd range : ... -> CS
'\xHH'-'\xHH' -> CSn
'\xHH'-'\xHH': '\xHH'-'\xHH' -> CSn
'\xHH'-'\xHH': '\xHH'-'\xHH' -> CSn
        ...
'\xHH'-'\xHH' -> CSn
}

escseqinfo{    # エスケープシーケンスのリスト（ステートフルの場合のみ）
#コードセット エスケープシーケンスの長さ エスケープシーケンス文字列
CS0      3      "\xHH\xHH\xHH"
CS1      3      "\xHH\xHH\xHH"
        ...
CSn      n      "\xHH\xHH\xHH..."       # x = N_CS 個まで
}

csinfo{ #コードセット情報
# MBLEN:エスケープシーケンスを除いたマルチバイト文字の長さ
# CS   MBLEN    第 1 バイト範囲     第 2 バイト範囲   ...   第 n バイト範囲
CS0     1       '\xHH'-'\xHH'
CS1     2       '\xHH'-'\xHH'     '\xHH'-'\xHH'
        ...
CSn     n       '\xHH'-'\xHH'     '\xHH'-'\xHH'       '\xHH'-'\xHH'
}

wcinfo{  #ワイド文字グループ情報
# グループ          ワイド文字範囲
GROUP0          L'\xHHHHHHHH' - L'\xHHHHHHHH'
GROUP1          L'\xHHHHHHHH' - L'\xHHHHHHHH'
        ....
GROUPn          L'\xHHHHHHHH' - L'\xHHHHHHHH'
#N_WCGROP 個まで
}

wctypeinfo{ # ワイド文字分類情報
#属性名       ワイド文字範囲 1                ワイド文字範囲 2              ...
alnum   L'\x00000030' - L'\x00000039'   L'\x00000041' - L'\x0000005a'
        L'\x00000061' - L'\x0000007a'
alpha   L'\x00000041' - L'\x0000005a'   L'\x00000061' - L'\x0000007a'
cntrl   L'\x00000000' - L'\x0000001f'   L'\x0000007f' - L'\x0000007f'
        ...
分類名    L'\xHHHHHHHH' - L'\xHHHHHHHH'   L'\xHHHHHHHH' - L'\xHHHHHHHH' ...
# N_TYPE 個まで
}

wctransinfo{ # ワイド文字変換情報
#属性名          範囲 1 : 変換先 1                          範囲 2 : 変換先 2      ...
tolower L'\x00000041' - L'\x0000005a' : L'\x00000061'
toupper L'\x00000061' - L'\x0000007a' : L'\x00000041'
        ...
分類名  L'\xHHHHHHHH' - L'\xHHHHHHHH' : L'\xHHHHHHHH' ...
```

```
# N_TRANS 個まで
}

mbtowc_cs0{ # コードセット 0 のマルチバイト/ワイド文字対応表
#マルチバイト文字　ワイド文字
"\x00"  L'\x00000000'
"\x01"  L'\x00000001'
"\x02"  L'\x00000002'
    ...
"\xHH"  L'\xHHHHHHHH'
}

mbtowc_csN{ # コードセット N のマルチバイト/ワイド文字対応表
#マルチバイト文字　ワイド文字
"\xHH\xHH..."   L'\xHHHHHHHH'
"\xHH\xHH..."   L'\xHHHHHHHH'
    ...
"\xHH\xHH..."   L'\xHHHHHHHH'
}

wctomb{ # ワイド→マルチバイト文字対応表
# マルチバイト文字とワイド文字が 1 対 1 対応ではなく、あるマルチバイト文字
# に対応するワイド文字が複数あり、ワイド→マルチバイト文字変換のときは
# その変換も行ないたいときはここに記述する
# ワイド文字     マルチバイト文字          コードセット
L'\xHHHHHHHH'   "\xHH\xHH..."           CSn
        ...
}
```

たとえば ja_JP.eucJP(日本語 EUC ロケール) の定義ファイルは一部次のようになっている。

```
# ja_JP.eucJP
generalinfo{
ENCODING STATELESS
MB_CUR_MAX 3
N_CS 4
N_WCGROUP 12
N_TYPE 11
N_TRANS 2
}
csjudgeinfo{
#1st range : 2nd range : 3rd range : ... -> CS
'\x00'-'\x7e' -> CS0
'\xa1'-'\xa8' -> CS1
'\xb0'-'\xf4' -> CS1
'\x8e'-'\x8e' -> CS2
'\x8f'-'\x8f' -> CS3
}
csinfo{
# codeset info
# CS   MBLEN    1st range    2nd range    3rd range
CS0 1 '\x00'-'\x7e'
CS1 2 '\xa1'-'\xf4'    '\xa1'-'\xfe'
CS2 2 '\x8e'-'\x8e'    '\xa1'-'\xdf'
```

```
CS3 3 '\x8f'-'\x8f'    '\xa1'-'\xed'    '\xa1'-'\xfe'
}
wcinfo{
# wc info
# GROUP         RANGE
GROUP0 L'\x00000000' - L'\x000000f7' # ASCII & latin-1
GROUP1    L'\x00024e00' - L'\x00029fa5' # Kanji
GROUP2 L'\x00000401' - L'\x0000045f' # Cyrillic
GROUP3 L'\x00000391' - L'\x000003c9' # Greek
GROUP4 L'\x00002010' - L'\x0000203e' # marks
GROUP5 L'\x00002103' - L'\x00002103' # marks
GROUP6 L'\x00002190' - L'\x00002193' # marks
GROUP7 L'\x000021d2' - L'\x00002312' # marks
GROUP8 L'\x00002500' - L'\x0000274d' # marks
GROUP9 L'\x00003000' - L'\x000030fe' # marks & Hiragana & katakana
GROUP10 L'\x0000ff01' - L'\x0000ff9f' # Fullwidth & Halfwidth
GROUP11 L'\x000002b9' - L'\x000002ba' # letter prime & double prime
}
wctypeinfo{
# wctype info
# PROPERTY RANGE1                RANGE2            ...
alnum L'\x00000030' - L'\x00000039' L'\x00000041' - L'\x0000005a'
L'\x00000061' - L'\x0000007a'
alpha   L'\x00000041' - L'\x0000005a' L'\x00000061' - L'\x0000007a'
cntrl L'\x00000000' - L'\x0000001f' L'\x0000007f' - L'\x0000007f'
digit L'\x00000030' - L'\x00000039'
graph L'\x00000021' - L'\x0000007e'
lower L'\x00000061' - L'\x0000007e'
print L'\x00000020' - L'\x0000007e'
punct L'\x00000021' - L'\x0000002f' L'\x0000003a' - L'\x00000040'
L'\x0000005b' - L'\x00000060' L'\x0000007b' - L'\x0000007e'
space L'\x00000009' - L'\x0000000d' L'\x00000020' - L'\x00000020'
upper L'\x00000041' - L'\x0000005a'
xdigit L'\x00000030' - L'\x00000039' L'\x00000041' - L'\x00000046'
L'\x00000061' - L'\x00000066'
}
wctransinfo{
# wctrans info
# PROPERTY  RANGE1 : MAPTO1        RANGE2 : MAPTO2     ...
tolower L'\x00000041' - L'\x0000005a' : L'\x00000061'
toupper L'\x00000061' - L'\x0000007a' : L'\x00000041'
}
mbtowc_cs0{
#mb  wc
"\x00" L'\x00000000'
"\x01" L'\x00000001'
"\x02" L'\x00000002'
"\x03" L'\x00000003'
 ... <中略>
"\x7d" L'\x0000007d'
"\x7e" L'\x0000007e'
}
```

```
mbtowc_cs1{
"\xa1\xa1" L'\x00003000' # zenkaku space
"\xa1\xa2" L'\x00003001' # zenkaku 'ten'
 ... <中略>
"\xf4\xa6" L'\x00027199'
}

mbtowc_cs2{
...<略>
}

mbtowc_cs3{
...<略>
}

wctomb{
# wc  mb cs
L'\x0000ff40' "\xa1\xae" CS1
}
```

マルチバイト/ワイド文字対応表のセクションではマルチバイト文字とワイド文字の対応を一つ一つ記述するのでかなり多くのエントリが並ぶ。ja_JP.eucJP(日本語 EUC ロケール) の定義ファイルの大きさはコメントも含めて 350k バイトほどになった。setlocale 関数でこのテキストファイルを読み込み、解釈してロケールデータ構造に収めるのにはかなり時間がかかる。それでこの定義ファイルを読み込んでバイナリ形式に変換するユーティリティプログラム mklcctype を用意した。このプログラムを用いてバイナリデータに変換したものを規定のディレクトリ (/usr/local/lib/ロケール名 または 環境変数${LOCALEDBPATH}/ロケール名) の下の LC_CTYPE という名前で保存する。setlocale はこのバイナリデータを読み込むことにした。このためデータ読み込みは高速化できたが、データがアーキテクチャ依存になってしまった。

第 5 章

評価

5.1 対応できたマルチバイト文字コード

いまのところロケールデータファイルが完成しているのは

- ja_JP.jis7... 日本語 7 ビット JIS
- ja_JP.eucjp... 日本語 EUC
- zh_CN.gb2312... 中国語 GB2312
- zh_CN.eucch... 中国語 EUC
- ko_KR.ksc5601... 韓国語 KSC5601
- ko_KR.euckr... 韓国語 EUC
- ja_JP.utf2... UTF-2(もちろん日本語専用というわけではない)

それとまだ不完全ながら CompoundText を使うロケールデータもある。

それぞれのロケールデータファイルのサイズは以下のとおりである。バイナリはロケールデータのメモリイメージのダンプなのでこれがこのままメモリ上でのサイズとなる。

	テキスト (コメント含む)	バイナリ
ja_JP.jis7	329677	281446
ja_JP.eucjp	353192	283438
zh_CN.gb2312	263038	300760
zh_CN.eucch	263072	302776
ko_kr.ksc5601	361451	328474
ko_kr.euckr	361500	330490
ja_JP.utf2	929942	464562

ISO2022-INT-1 のロケールを完全に実現することはこのライブラリのワイド文字では難しいことが分かった。ISO2022-INT-1 は以下の文字集合をサポートしている。

文字集合	エスケープシーケンス
ASCII	ESC (B
JISX0201 ローマ字	ESC (J
ISO8859-1	ESC . A
ISO8859-7	ESC . F
JISX0208-1978 日本語	ESC $ @
JISX0208-1983 日本語	ESC $ B
GB2312-1980 中国語	ESC $ A
KSC5601-1987 韓国語	ESC $ (C
JISX0212-1990 日本語	ESC $ (D
CNS11643-1986-1 中国語	ESC $ (G
CNS11643-1986-2 中国語	ESC $ (H

このライブラリのワイド文字コードでは日中韓の判別はできるがもとのマルチバイト文字コードが日本語の何コードか、中国語の何コードかといった判別はできない。同じ日本語のコードでも JISX0208 と JISX0212 のように全く違う文字集合であればよいが、GB2312 と CNS11643 のように同じ文字が両方のコードに存在する場合、いったんマルチバイト→ワイド文字変換するとワイド→マルチバイト文字変換のときどちらのコードにすればよいか分からなくなってしまう。

5.2 対応できたシステム

現在のところ BSD/386 1.0 のみで利用できる。他のシステムでもコンパイルはできるが正常に動作しない。とくに mklcctype コマンドによるロケールデータの生成と setlocale 関数のロケールデータのロードの部分が正常に動作していない。エンディアンやパディングの影響と考えられるが詳しい原因については調査中である。

5.3 メモリ消費について

BSD/386 の場合、ロケールデータの保持に必要なメモリ容量は以下のようになった。

	ja_JP.jis7	ja_JP.eucjp	ja_JP.utf2
コードセット判別テーブル	64	2048	6144
マルチバイト→ワイド文字変換テーブル	53668	53676	132152
ワイド→マルチバイト文字変換テーブル	227002	227002	325434
コードセット情報テーブル	96	96	216
ワイド文字分類テーブル	476	476	476
ワイド文字変換テーブル	88	88	88
_lcctypeinfo 構造体	52	52	52
合計	281446	283438	464562

マルチバイト→ワイド文字変換テーブルはほぼ設計通りの大きさだが、ワイド→マルチバイト文字変換テーブルが非常に大きくなってしまった。ja_JP.eucjp ロケールのワイド→マルチバイト文字変換テーブルを例にとると、各ワイド文字グループごとにグループ判別テーブルの内訳は

グループ	主な文字集合	サイズ
0	ASCII & latin-1	2496
1	漢字 (JISX0208 & JISX0212)	209036
2	キリル文字 (JISX0208)	966
3	ギリシャ文字 (JISX0208)	586
4	記号 (JISX0208)	486
5	記号 (JISX0208)	26
6	記号 (JISX0208)	56
7	記号 (JISX0208)	3226
8	記号 (JISX0208)	5916
9	記号、ひらがな、カタカナ (JISX0208)	2566
10	半角カナ (JISX0201)、全角アルファベット (JISX0208)	1606
11	記号 (JISX0208)	36

となる。漢字に対応するテーブルが大部分を占めているが、これは文字数が多いだけでなく、ワイド文字コードである ICODE はもともと Unicode の日中韓の漢字を統合したコードに日中韓の識別のフラグをつけたものなので、日本の漢字にはなくても中国や韓国の漢字には存在する文字に割り当てられているコードも少なくないためこのグループ内に文字の割り当てられていないコードが不規則にかなり多く存在するためである。また変換テーブルのエントリサイズを固定 (コードセット 2 バイト (short 型) とマルチバイト文字 8 バイト (char の配列)) にしたため、実際には漢字の場合マルチバイト文字には 2 バイトしか使わないのに無駄が多いという点もある。

5.4 処理速度について

5.4.1 マルチバイト↔ワイド文字変換

ロケールデータのロードの部分が BSD/386 以外のシステムではうまく動作しなかったので日本語 EUC のロケールデータをプログラム中に書き込みコンパイルした。それでマルチバイト文字→ワイド文字→マルチバイト文字変換をする図 5.1 のようなプログラムを書いてシステムの標準ライブラリ関数を用いた同様の動作をするプログラムと実行速度を比較した。

すべて gcc(-O2 オプション) でコンパイルし、これに 5000 行の日本語 EUC のテキストを読みこませて time コマンドで実行にかかる時間を計測した。結果 (何回か実行した時の user time の最良値) は以下のとおりである。

- 本ライブラリを用いたプログラム

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>

main(){
    wchar_t ws[BUFSIZ], *wptr;
    char    buf[BUFSIZ], *ptr;

    setlocale(LC_ALL, "");
    while(fgets(buf, BUFSIZ, stdin) != NULL){
        ptr = buf;
        mbsrtowcs(ws, &ptr, BUFSIZ, NULL);
        wptr = ws;
        wcsrtombs(buf, &wptr, BUFSIZ, NULL);
    }
}
```

- システムの標準ライブラリ関数を用いたプログラム

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

main(){
    wchar_t ws[BUFSIZ], *wptr;
    char    buf[BUFSIZ], *ptr;

    setlocale(LC_ALL, "");
    while(fgets(buf, BUFSIZ, stdin) != NULL){
        ptr = buf;
        mbstowcs(ws, ptr, BUFSIZ);
        wptr = ws;
        wcstombs(buf, wptr, BUFSIZ);
    }
}
```

図 5.1: マルチバイト↔ワイド文字変換テストプログラム

	本ライブラリ	システム付属のライブラリ
HP-UX	0.76	0.29
NEWS-OS	1.57	1.04

単位:秒

システム標準の関数を用いたものよりかなり遅い。これはシステム標準関数はワイド文字も EUC に少し手を加えた程度のものでマルチバイト文字との変換にほとんど処理が必要ないためだと思われる。本ライブラリはさまざまなマルチバイト文字コードに柔軟に対応できることに主眼がおかれていたため変換速度が犠牲になっている。

5.4.2 ロケールデータのロード

上記のこのライブラリを使ったプログラムを今度はロケールデータをきちんとファイルから読むようにしてコンパイルし、プロファイラで各関数の実行時間を見てみると以下のようになった (入力ファイルは 3000 行の日本語 EUC テキスト)。

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 34.5      1.62     1.62                             mcount (21)
 19.6      2.54     0.92   156000     0.01     0.01  __mbtowc_trans [6]
 10.9      3.05     0.51   153000     0.00     0.01  __wcrtomb_sl [7]
  9.0      3.47     0.42       30    14.00    14.00  _read [9]
  6.8      3.79     0.32   156000     0.00     0.01  __mbrtowc_sl [4]
  5.5      4.05     0.26   153000     0.00     0.00  _strncpy [11]
  5.3      4.30     0.25     3000     0.08     0.50  _usrreq_mbsrtowcs [3]
  5.1      4.54     0.24     3000     0.08     0.34  _usrreq_wcsrtombs [5]
  0.9      4.58     0.04     3026     0.01     0.01  _bcopy [13]
  0.9      4.62     0.04     3001     0.01     0.17  _fgets [8]
  0.4      4.64     0.02     3026     0.01     0.01  _memchr [15]
  0.4      4.66     0.02        1    20.00    20.00  _calloc [16]
  0.2      4.67     0.01       29     0.34    14.34  ___srefill [10]
  0.2      4.68     0.01        1    10.00    10.00  _close [17]
  0.2      4.69     0.01        1    10.00    10.00  _open [18]
  0.0      4.69     0.00      153     0.00     0.00  _strncmp [22]
  0.0      4.69     0.00       29     0.00     0.00  ___sread [23]
 <後略>
```

read システムコールは呼ばれる回数が少ないにもかかわらずかなりの時間がかかっている。これはロケールデータのロードに read システムコールを使っているためと思われる。

第 6 章

結論

6.1 このライブラリの利点

文字コードの違いを意識しないでプログラミングができるようにするという目的はほぼ達成することができた。たとえば文字数をカウントするプログラムはロケールデータが用意されているコードであれば、以下のように標準ライブラリ関数を用いてバイト数をカウントする場合とほとんどおなじ感覚で記述できる。

```
#include <locale.h>
#include <wchar.h>
#include <wctype.h>
#include <stdio.h>

main()
{
    wchar_t wcs[BUFSIZ];
    size_t count = 0;

    setlocale(LC_ALL, "");
    while(fgetws(wcs, BUFSIZ, stdin) != NULL)
        count += wcslen(wcs);
    printf("%d\n", count);
}
```

実行前に環境変数 LANG(もしくは LC_CTYPE, LC_ALL) にロケール名をセットしておけばその文字コードに対応した文字数カウントプログラムとなる。

またワイド文字コードを ICODE に統一し、マルチバイト/ワイド文字変換のアルゴリズムを工夫することによって多くの文字コードに対応する柔軟性を持たせることができた。

6.2 このライブラリの問題点

マルチバイト/ワイド文字変換の柔軟性に重きをおいていたため、変換スピードとメモリ消費量を減らすことが犠牲になっている。これが許容できる範囲でバランスがとれているとは言い難い。スピード、メモリの節約のためのチューニングが必要である。ロケールデータのロードに時間がかかるのはロケールデータファイルが大きいこともあるがデータ構造が複雑なため、この構造になるようにさまざまなところからポインタをつけてやる必要があることも関係しているだろう。もっと単純なデータ構造での同等の柔軟性を実現できるように改良したい。

また実装を進めていくうちにこのワイド文字の設計では対応できないコードがあることにも気づいた。これについてはワイド文字コードの使用の見直しが必要であるが、言語と文字コードとはどのような関係にあるかをよく考慮し直してみる必要があると感じた。たとえば中国本土でひろく用いられている GB2312 と台湾で用いられている CNS11643 はどちらも中国語用のコードであるが、コード体系はかなり異なる。ICODE の元となった Unicode はこれら 2 つをふくめ漢字のコードをすべてを統合しているがこのような考え方は正しいのどうか、それともそれぞれべつのコードと考えるのがよいのかも検討を要する問題である。

第 7 章

今後の課題

7.1 よりよいワイド文字の設計

いまのワイド文字の設計では対応しきれない文字コードがあること、また日本語、中国語、韓国語を扱う場合、無駄が多いことを考えるとワイド文字コードの改良が必要である。また結合文字への対応も考慮したい。

7.2 アプリケーションへの適用

7.2.1 コンパイラへの適用

プログラム中にワイド文字定数を書きたいこともしばしばあることだろう。ANSI C ではワイド文字定数を

```
wchar_t wc = L'あ';
wchar_t wcs[] = L"あいうえお";
```

のように記述できることになっている。もちろんソースプログラム内にはマルチバイト文字で書かれているが、コンパイラはコンパイル時にこれをワイド文字に変換しなければならない。それでコンパイラも他言語対応になる必要がある。

7.2.2 UNIX の文字処理ユーティリティの多言語化

sed, awk, wc などをこのライブラリを用いて書き直し、多言語対応にする。

7.2.3 高レベルライブラリの多言語化

国際化プログラミングを容易にするためには Curses や regex、dbm などのより高いレベルのライブラリの国際化も必要である。

謝辞

指導をして下さった村井先生、徳田先生に感謝する。また最新の MSE の仕様書、ISO10646 の仕様書など必要な資料を提供して下さった斎藤先生に感謝する。そして実装についてさまざまなアドバイスを与えて下さった JUS の usrreq 研究会のメンバに感謝する。

参考文献

[1] 日経エレクトロニクス 1992/4/27、日経 BP 社

[2] Rob Pike & Ken Thompson, Hello World or Καλημερα κοσμε or こんにちは世界

[3] SoftwareDesign 1993/3 、技術評論社

[4] 日本サン ・マイクロシステムズ JLE プログラマーズガイド

[5] UNIX システムラボラトリーズパシフィック、UNIX SystemV リリース 4.1 拡張セキュリティ プログラマ ・ガイド¡国際化機能¿、共立出版、1992

[6] P.J.Plauger, THE STANDARD C LIBRARY, PRENTICE HALL, 1992

[7] HEWLETT PACKARD, HP-UX リファレンス (リリース 9.0)Vol.2,1992

[8] HEWLETT PACKARD, Native Language Support:User's Guide, 1992

[9] Draft International Standard ISO/IEC DIS 10646-1.2,1992

[10] 月刊スーパーアスキー 1991/2、アスキー

[11] The Unicode Consortium, The Unicode Standard *Worldwide Character Encoding* Version1.0, Addison Wesley, 1991

[12] ISO/IEC JTC1/SC22, Proposed Draft Amendment 1 to ISO/IEC 9899:1990 Programming language C on: Normative Integrity Addendum, August 1993

[13] Robert W. Scheifler, Compound Text Encoding Version 1.1, MIT X Consortium Standard X Version 11, Release 5, 1989

[14] 太田昌孝、ＩＳＯ１０６４６と国際化された文字コードについて、 Proceding of JWCC'93 in taiwan