文字コードの生い立ち
～ グローバル化における先人の知恵 ～

8/25/2011
鵤 由利

はじめに

かつて、IT 業界において日本が世界に向かって大きく発言した時期があった。そこでは、国際化という名のもとに、コンピュータで扱うことのできる文字というものに対して世界と協調しながら進めていき、相応の形にしていった。ある時は、世界を相手に大いに議論し、ある時は、日本国内でさまざまな議論を繰り返した先人たち。結果に関しては、いろいろと意見があり賛否両論分かれるところも残ってはいるが、ブラウザではいろいろな国の文章を見ることができるし、環境設定さえすればどこの国でも使用することができるソフトウエアが作られている。このような観点から見れば、その功績は大きいものといえる。
ここでは、文字コードというものがどのように変化し、現在の形になっていったかということに関して、その変遷を記述していきたい。

1. コンピュータの文字は何ビット？

まず、「コンピュータで文字を扱うとすると何ビット必要なのか？」ということを出発点として話を進めていく。メモリーやディスク、はたまたプロセッサーが小さい時代では、文字を最小限のビット数で表現するということが重要であった。また、その後のテクノロジーの進化の過程にあって、都度、「文字を扱うのに何ビット必要なのか？」というのが根本にある考え方であった。
まず、アルファベットを表現するのに何ビット必要なのであろうか。アルファベットが 26 文字、数字が 10 文字、合わせて 36 文字が最低限必要である。5 ビットでは 32 文字までということになるから、英数字を表現するには 6 ビット必要になる。6 ビットであれば、64 文字まで表現できるので、アルファベット、数字、それといくつかの制御文字（スペース、改行など）を含めることができる。昔の通信システムでは、この 6 ビット（あるいは 5 ビットでアルファベットと数字を切り替えながら使う）というのがあったようである。
その後、ASCII コードが定義され、コンピュータのコード体系としての基本が出来るわけであるが、この ASCII では 7 ビットが使われ、アルファベットの大文字・小文字、数字、コントロール・コードを含め 128 文字の中に収められることになった。そのほか、メインフレーム系を中心に EBCDIC コードというものも定義され、こちらは 8 ビットを使ったコード系となっているが、ここではその内容は割愛させていただく。
さて、ASCII が定義されて以降は、コンピュータの処理能力の向上に伴ってアルファベットだけでなく各国の言語にコードを割付けて処理できるようにしようという流れができたわけであるが、その拡張の段階において、思惑、過去のしがらみのため、様々な変遷をたどることになる。ここでは、細かいコード系の説明は行なわないが、その背景となる内容をいくつか触れてみたいと思う。
まず、ASCII が 7 ビットで定義されている。しかしながら、2 進数でできているコンピュータの表現では 7 ビットではなく 8 ビットで処理を行う方が都合が良い。いわゆるバイトという概念である。そうすると、余った 1 ビットを何に使おうかということを考えてしまうことになる。この時点では、まだ、アルファベット以外の文字を処理しようという考えはないので、この 1 ビットをプログラムの処理に使ってしまうという発想になるわけである。1 ビットといえども馬鹿にならない。フラグとして使うことで、いわゆるバイト列に対してマーカーとして意味を持たせることができ、プログラムの処理を簡単化することができる。反面、これは、その後の拡張性という点で問題を抱えることとなってしまうのである。
このようなプログラミングの流れと平行して、この 1 ビットを使って英語以外の言語を扱えるようにしようという考えが起こる。つまり、7 ビットで扱える文字数は 128 文字までであるが、これを 8 ビット使えるようにすれば、256 文字扱えることになる。英語以外にもう 1 つの言語（文字コード）を表現できるようになるわけである。これを日本語に当てはめると、対象になるのがカタカナである。カタカナ、

濁点、半濁点等を合わせても 128 文字以内で収まるわけであるから、英語とカタカナという文字コードが 8 ビットを用いることによって表現できることになる。
さて、そうすると、上記のプログラミングの処理と文字コードの拡張という点において、1 ビットの扱いがぶつかってしまうことになる。せっかくカタカナのみとはいえ日本語が扱えるコード体系ができても、肝心のプログラムで使用することができないということになってしまう。このプログラムの問題を解消するため、いわゆる「8 ビット・クリーン・アップ」ということで、どれだけの開発者の工数が取られたかは、80 年代後半を生きたプログラマーには懐かしい話である。
話はずれるが、7 ビットから 8 ビットというのは、実際にはコンピュータの処理能力が上がったわけではないが、最後の 1 つのビットに対する考え方で、将来の拡張性（ここでは他国語言語処理）というものに影響を及ぼしてしまったことになる。「どこまで先を読んだプログラミングができるか」、また、「現状のアーキテクチャの制限の中からどこまで拡張性のあるプログラムができるのか」。これは、プログラマーにとっては大きな挑戦ということになる。

2. 漢字コード体系の変遷

さて、次に 8 ビットから先、どのように漢字体系が出来上がってきたか、その変遷に関して触れてみたいと思う。
8 ビットで入りきらない漢字を入れるわけだから、次は 16 ビット。つまり、2 バイトを使うということになる。2 バイトを単純に考えると、65,536 文字まで定義することができる。とりあえず日本で普通に使われる漢字から定義していくと、いわゆる JIS の第 1 水準、第 2 水準、第 3 水準ということになっていく。まずは、第 1 水準、第 2 水準から始まるが、第 1 水準漢字で約 3,000 文字、第 2 水準漢字で約 3,400 文字。非漢字を合わせても 7,000 文字程度である。
ここで、コード系との絡みが出てくるわけであるが、前の章でも触れているが、言語系を 7 ビットの集合で表現し、それを 2 枚合わせて 8 ビット体系としている。詳しくは、ISO2022 のコード拡張方法となるのであるが、要は 7 ビットの言語の集まりの箱を組み合わせ、出し入れしていろいろな言語を表現しようということである。では、2 バイトを 7 ビットx7 ビットで表現するとどうなるか。表現できる文字数は、128x128=16,384 文字となる。実際には、制御文字の領域を除いた 94x94=8,836 文字がそのエリアとなる。
JIS では、非漢字（記号や数字、アルファベット、ひらがな、カタカナ）、第 1 水準漢字、第 2 水準漢字合わせて約 7,000 字を定義し、これを用いることとした。その後、第 3 水準などを別のプレーンとして定義している。日常使用しているというレベルでは、ほぼ第 1 水準で十分であるし、第 2 水準の漢字はあまり読めないし使わない漢字がほとんどである。
さて、ここでコード系の話に戻るが、この漢字用の JIS の定義ができたことにより、これをどのように使用していくかということで、いままでのしがらみ、国際標準との整合性、はたまた、新たな流れ（Unicode で代表される）ということが起こってくるわけである。
まず、2 つの文字セットのうち 1 つは ASCII で決まりである。これは、既存ソフトウエアとの互換性を考えるとそうならざるをえない。なにせ、いたるところで ASCII のコードが直接プログラムされているし、OS ですら、ディレクトリやフォルダーにはバックスラッシュ（＼）が使われていたりするように、ASCII を前提とせざるをえない。なお、日本語 Windows ではフォルダーの区切りは￥じゃないか、という人がいるかもしれないが、プログラムの内部的には￥とバックスラッシュは同じコードであるために、画面上￥となっているにすぎない。ここは、JIS ローマ字という ISO646 の派生系が残ってしまったための悪しき流れであるが、ここではこれに関しては詳しくは述べない。
さて、あと 1 つの文字セットとしてこの JIS 漢字を入れれば、非漢字、第 1 水準漢字、第 2 水準漢字、それと余ったところに適当に勝手な文字（たとえば絵文字　;-)）を入れれば事足りることになりそうである。しかしながら、ここで過去のしがらみが発生する。この 2 つ目の文字セットには、すでにカタカナ文字セットが使われてしまっている。ということは、簡単に JIS 漢字をあてはめることができない。
さてさて、ここでシフト JIS というコードが出てくるわけであるが、知恵を絞ればそれなりに解決策というものが出てくるようである。まず 7 ビットで表現できるコードは 128 文字であるが、実際には 94 文字分しか使用していない。後は、コントロール・コードとして使用している。また、カタカナで使用している文字数は、実際には 64 文字である。ということは、カタカナに使用しているところ以外に 64 文字分の空きスペースがあると考えることができる。そこに、漢字を割り当て、第 1 バイトをこの空きスペースを使用すればよい。第 2 バイトは、8 ビット全般にわたって使えると考えれば、これで漢字を入れ

るのに十分なスペースが確保できてしまった。Windows で広く使われている MS 漢字あるいは CP932 は、このシフト JIS をベースにしており、完全に市民権を得てしまった。
適当に、隙間を狙って作ってしまったようなシフト JIS コードであるが、実は、ある意味ではよくできているコード体系である。今と違って、当時は文字端末。グラフィック端末が主流になる前である。この文字端末の世界では、画面上の文字幅とバイト幅が一致している必要があった。逆に言うと、一致していないと処理がぐちゃぐちゃになってしまう。このシフト JIS というコードは、1 バイトの英数字とカタカナは 1 文字分の幅、2 バイトの漢字は 2 倍の幅ということで、完全に文字幅とバイト幅が一致し、プログラム開発者には非常にありがたいコード体系となっていた。ちなみに、半角カタカナとか全角カタカナ（JIS 漢字の非漢字部分のカタカナ）というような日常離れした文字表現が出てきたのも、このころのコード体系に基づいたものである。

3. 国際標準への適合

さて、このようなシフト JIS に対して、やはり標準からの乖離が問題となってくる。特に国際化の波の中では、日本語だけがなんとか入れられればというのでは済まないわけで、コード体系としての国際化に準拠したコード系が必要になってくる。これが、いわゆる EUC(Extended UNIX code)というもので、UJIS とか EUC-JP とか言われているものである。こちらは、UNIX を中心に展開されてくることになる。
さて、EUC というコード体系であるが、これは国際標準の ISO-2022 の文字コードの拡張方法に準拠した形で作られている。今まで述べてきたように、8 ビットの文字体系では 2 つの文字セットを表現することができる。EUC では4つの文字セットを扱い、デフォルトで呼び出されている 2 つの文字セットと、別途呼び出し用のコード(シングルシフト)を用いて残りの 2 つの文字セットを呼び出すことで実現している。EUC の特徴は、1つ目の文字セットを ASCII に決定し、残りの 3 つの文字セットを呼び出す方法を取っていることである。これにより、既存のソフトウエア（ASCII ベース）との互換性を維持できるようにしている。

この EUC のスキームに基づいて、日本語 EUC では以下の構成を取った。ここで、G0,G1,G2,G3 というコード系で使われる名称を使用するが、上記の 4 つの文字セットに関するそれぞれの箱の名称と捕らえていただければ良いと思う。

G0: ASCII （JIS ローマ字）
G1: JIS X0208 (非漢字、第 1 水準漢字、第 2 水準漢字)
G2: JIS X0201(カタカナ)
G3: JISX0212(補助漢字) あるいは、ユーザ定義文字

G0 と G1 の ASCII および JIS X0208 漢字をデフォルトとして使用し、JIS カタカナおよび補助漢字をシングルシフトで呼び出すという体系である。なお、G0 にある ASCII と JIS ローマ字文字セットの絡みの件は前の章で触れた内容になるので、ここでは割愛する。

この日本語 EUC のコード体系をプログラミングの観点から見ると以下のことが言える。まず、国際標準に準拠しているため、プログラム自体は基本的に全世界で共通に使えるものにすることができる。しかしながら、シフト JIS の時に述べたような「文字幅とバイト数が一致」ということに関しては、残念ながら実現できないこととなってしまっている。これは、G2 および G3 の文字セットを使用する場合にはシングルシフト・コードが必要となるため、たとえば JIS カタカナを使用する場合には、1 文字に対してシングルシフト・コードとカタカナ・コードの 2 バイトが必要となってしまうからである。また、既存（ASCII とカタカナの時代）のコード系とは互換性がないことになる。日本語だけを処理する場合を考えると、シフト JIS コードの方が優れていると言えるかもしれない。

ここで、「文字幅」と「バイト数」の関係というものを別の角度から考えてみたい。今まで述べてきた「文字幅とバイト数が一致」ということが本当にプログラム上処理しやすいのであろうか？確かに、バイト順に並べていくものと画面上の文字の並びが一致するため処理がしやすいという点はある。しかしながら、漢字の場合、1 文字を 2 バイトで表現しているにも関わらず、内部処理は 1 バイト単位に行われるための問題が発生する可能性がある。たとえば、ある 1 バイトを取ってきて、それが、漢字の 1 バイト目なのか 2 バイト目なのかを判別するのは非常に困難である。いわゆる、ずれマッチ（漢字の 2 バイト目と次の漢字の 1 バイト目を合わせて別の漢字と認識してしまうこと）を引き起こすことになる。そうすると、さらにプログラムにとって理想的なコード系にするためには、文字の長さが同一、

つまり、文字単位で処理を行うことができるようにしたい。EUC では、上記のコード（外部コードと呼ぶことがある）に対して、内部コードを定義している。これは、2 バイト固定長として、上記 G0 から G3 までの 4 つの文字セットを表現するものである。各文字セットは 7 ビットで表現されているので、2 バイト空間として以下の 4 つの構成で表現できることになる。日本語 EUC を当てはめてみると以下のようになる。（xxxxxxx には、実際の 7 ビットのコードが入る）。

00000000 0xxxxxxx : G0 (ASCII)
1xxxxxxx 1xxxxxxx : G1(JIS X0208)
00000000 1xxxxxxx : G2 (JIS カタカナ)
1xxxxxxx 0xxxxxxx : G3(JIS X0212)

ちなみに、前者は情報交換用あるいはファイル・コード、後者は内部処理コードと呼ぶ場合もある。
今までの説明から EUC の外部コード、および内部コードの特徴を述べると以下のようになる。

- 外部コード
  バイト列処理（ストリーム処理）が可能。バイト単位の処理であり、複数バイト文字のための特別な処理が必要。
- 内部コード
  1 文字ごとの長さが同じ。バイト単位ではなく文字ごとの処理が可能。バイト列の中に NULL コードや制御コードが含まれるため、バイト列処理は不可。

なにやらごちゃごちゃしてきてしまったが、既存のバイト単位の処理（文字幅とバイト数が一致）の環境に複数バイトを必要とする処理を持ち込んだために起こったことによる複雑さというものが、ここで発生してしまったことになる。8 ビット・クリーン・アップに続いて、マルチバイト処理、そして、いわゆる国際化というプログラミングが新たに必要となった。

ここまで来ると、次の展開は大体読めてくると思われるが、世界中全ての文字を 1 つの構造に入れてしまおうという試みについて次章で述べていく。

4. 国際化文字コードおよび Unicode

文字コードの変遷の最後として、国際化文字コードおよびその中での日本語のあり方の彷徨に焦点を当ててみる。前章の最後に述べた「世界中全ての文字を 1 つの構造に入れてしまおうという試み」に関して記述していきたい。
そもそも世界中の言語に番号を振っていったら、いくつまであれば足りるのであろうか？つまり、世界中の言語を入れるための器（コード体系）を作るとしたならば、何バイト必要となるのであろうか。本シリーズで順次述べてきたように、アルファベットと数字で 6 ビット、さらに進めて小文字、記号等を入れて 7 ビット、ヨーロッパ言語やカタカナを必要に応じて入れることで 8 ビット、つまり 1 バイト、漢字やいわゆるイデオグラフィック文字（象形文字）の基本セットを入れるのに 16 ビット、つまり 2 バイトということになる。それでは、まず 2 バイトということで始めてみる。いわゆるアルファベット（ASCII やヨーロッパ言語）を 2 バイトの中に全て入れることができる。漢字、中国語、韓国語の基本セットを入れることもできる。漢字で言えば、JIS-X0208 で定義されるところの第 1・第 2 水準である。しかしながら、漢字だけを取ってみても、大漢和辞典クラスで約 50,000 字。中国語、韓国語を含め全世界の文字に対してコードを振ろうとすると、2 バイトで表現できる文字数である 65,536 字では不可能である。ちなみに、TRON では、超漢字という名の下に、約 17 万字を定義しているようである。もちろん、2 バイトではなくそれ以上のバイト数が用いられている。
それでは、Unicode はどうかというと、2 バイトに収めるために、この問題に対して Unify という方法を取っている。日本語、中国語、韓国語などの漢字のうち、同じ（似通ったと言ったほうが良いかもしれない）文字に関しては同じコードを割り振り同一漢字とするというものである。漢字のルーツは同じであるので、漢字が最初に考案された時代まで遡れば正しいといえるであろう。若干の文字の違い、たとえば、点が1つ多いとか、線の角度が違うとか、線がくっついているとか離れているとかは、情報を処理するということから考えればたいした問題ではないかもしれない。一方、文字はたとえ元は同じ文字であったとしても、その後の過程の中でそれぞれの文化の中で変化してきているものである。その変化の過程を完全に明確にせずに一意のコードを振ることで十分なのかどうかという問題もある。

Unicode では、文字の違いを表現上の問題、いわゆるプリゼンテーション・フォームで吸収するという立場を取っている。リッチなドキュメントが主流となっている状況では、文字ごとに属性があるものとして別のコードを振るよりは、コードとしては同じで、表現上の属性を別途持たせるということで十分であるかもしれない。しかしながら、すべてのドキュメントに対して、書かれている文字を意識しながらプリゼンテーション・フォームを指定して作成するということは現実的には不可能であろう。
このような状況の中で、ISO10646 は制定され、簡単に言うと Unicode プラス4バイトへの拡張という形になった。グローバルに相互の通信ができるようになったという点では、Unicode および ISO10646 の価値は大きい。また、Unicode/ISO10646 は単に黒舟の来航というわけではなかった。この制定および日本での展開という点においては、多数の日本人が尽力された。国際会議において日本のポジションを主張し、成果をあげるべく努力され、また、日本での実際の運用に関してさまざまな団体で議論され実装できるようなガイド作りを行った。現在、この文字コードが幅広く使用されている状況を見ると、グローバル環境における日本の存在というものに関して、1 つの指針となる活動であったと思われる。

5. グローバル化に対する日本人の取り組み

最後に、以上のような文字コードの変遷を参考にしながら、日本人としてどのようにグローバル化に取り組んでいくべきか、その指針として以下の 2 点を上げたい。

① 既存のものを変える勇気を持つ
決してローカルな基準や既成事実のようなものにとらわれることなく、柔軟に対応していくことが重要である。また、たとえ一時的な不利益が発生しようとも、変えることに対する勇気を持つことが大切である。上記で記述したように、フォルダーの区切り記号として￥を使い続けざるを得なくなってしまった状況、アルファベットを用いたキー入力しか行わない人でも 106 キーボード（日本語キーボード）を使い続けざるをえない状況のように、国際的な流れから取り残されていくことは是非とも避けていきたいものである。

② 積極的にグローバルの活動に参画し、国際標準（デファクト・スタンダードも含む）への準拠を進める
グローバルに行われている活動に対して積極的に参画し、日本の立場・意見を述べていくことが重要である。「日本は別」的な発想では太刀打ちできないことは明らかである。それぞれの立場においてグローバルな視点を持ち、積極的にその活動に参画していくことが重要である。その意味では、Unicode や ISO10646 等で、当時の人々が国際会議に積極的に参加し、日本の立場を明確に主張し、限界ぎりぎりの議論を行い、また、標準化制定のために積極的に活動したことは大きな道標になるものと思われる。また、これらの標準が決まった後に、さらに日本での標準化および利用に関して活動し、「日本は別」というような状況にならないように努力された方々の活動も、同様に大きな道標にしていかなければならない。

[End]