Interchange of Non-English Computer Text

Frank da Cruz

Columbia University, New York
1994

Introduction

Thirty years ago, computers and people communicated using a small repertoire of symbols, digits, and Roman letters. Often the letters were only in uppercase, and there were no accents. The language of computing was exclusively English. Today, we are poised on the brink of a worldwide computer-based communications revolution, with a single character set encompassing all the world's writing systems.

In the intervening decades, we have contrived a vast Babel of mutually incomprehensible character sets, both proprietary and standard. The new Universal Character Set, ISO 10646 [19], offers a single common encoding for all writing systems. But radical and massive changes are required in data entry and display hardware as well as in computer software and data files at all levels from operating system to application, and could therefore take decades to see widespread use. In the meantime, how shall we survive our Babel?

The problem is immediately apparent to anyone who tries to transfer non-English textual data between two different kinds of computers using conventional methods. ''Pâté'' on an IBM PC becomes ''PÉtä'' on the Macintosh, and a truckload of pocket-bread is delivered instead of goose liver.

This paper presents a simplified and condensed description of the character-set translation method developed for the Kermit file transfer protocol. The lessons learned should be useful in any arena where text must be transmitted meaningfully between unlike computer systems. Familiarity with the standards process in general, and the US ASCII [1], ISO 646 [14], and ISO 8859 [18] character set standards is assumed, and with ISO Standards 2022 [15] and 4873 [17], as well as with proprietary character sets such as IBM PC code pages.

Types of Character Sets

Today's coded character sets can be classified along several axes: standard versus proprietary, 7-bit versus 8-bit, single-byte versus multibyte, and so on. This paper treats only the character sets used in application-independent plain-text files, and not the application-specific text representations used in word processing, publishing, and similar environments that are concerned with rendering forms such as fonts, style, point size, ligatures, and so forth, and which sometimes offer character repertoires different from any plain-text character set.

Standard Character Sets

Let us define a standard character set as one that is registered in the ISO *Register of Coded Characters to Be Used with Escape Sequences* [21] under the provisions of ISO Standard 2375 [16]. The Register, which is maintained by the European Computer Manufacturers Association (ECMA), includes listings of all ISO-registered character sets and assigns unique registration numbers and designating escape sequences to each one.

Standard character sets are subdivided into two major types: graphic and control. Thus ASCII, which is the USA version of ISO 646, and which most people think of as a single character set, is really two sets: the ISO 646 32-character control set (ISO registration number 001) and a 94-character graphics set (ISO registration 006). And, as specified by ISO Standard 4873, the characters Space and Delete are not part of ASCII *per se*, but rather separate components that must always be available in the presence of a 94-character graphics set.

Similarly, ISO 8859-1 Latin Alphabet 1, which most of us think of as a coherent 8-bit character set is, in truth, composed of the ISO 646 control set (registration 001), the ISO 646 USA graphics set (006), the characters Space and Delete, a second 32-character control set (normally, but not necessarily, ISO 6429 [20], registration 077), and a 96-character set known as ''The Right-hand Part of Latin Alphabet 1'' (registration 100). Each of these pieces except Space and Delete has its own unique registration number and designating escape sequence. There is no single, unique identifier for the 8-bit Latin-1 character set in its entirety.

```
   <--C0--> <---------GL----------><--C1--> <---------GR---------->
     00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15
   +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
00 |       | SP                    |       | _ |                   |
01 |       |                       |       |                       |
02 |       |                       |       |                       |
03 |   C   |       ASCII           |   C   |       Special         |
04 |   o   |       graphics        |   o   |       Graphics        |
05 |   n   |                       |   n   |                       |
06 |   t   |                       |   t   |                       |
07 |   r   |                       |   r   |                       |
08 |   o   |                       |   o   |                       |
09 |   l   |                       |   l   |                       |
10 |   s   |                       |   s   |                       |
11 |       |                       |       |                       |
12 |       |                       |       |                       |
13 |       |                       |       |                       |
14 |       |                       |       |                       |
15 |       |                    DEL|       |                   , - |
   +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
   <--C0--> <---------GL----------><--C1--> <---------GR---------->
```

Figure 1: Structure of a Standard 8-Bit Single-Byte Character Set

In practice, most standard character sets have the same structure, which is illustrated in Figure 1:

1. The 32-character control set of ISO 646 in columns 0 and 1. This is called the C0 region.
2. The character Space at position 2/0.
3. A 94-character graphics set in positions 2/1 through 7/14. This is called the Graphics Left, or GL, region. For 7-bit character sets, these are the 84 characters of the ISO 646 International Reference Version plus 10 country-specific characters. For 8-bit sets, these are the 94 graphic characters of US ASCII.
4. The character Delete at position 7/15.
5. For 8-bit sets, a 32-character "right half" control set in columns 8 and 9. This is the C1 region.
6. For 8-bit sets, a 94- or 96-character "right half" graphics set in columns 10 through 15. This is the Graphics Right or GR region.
7. In multibyte graphic sets (not illustrated in the figure), each character is composed of a fixed number of bytes, generally two, with each byte in the graphic range, i.e. not corresponding to a C0 or C1 control character, and following either the 94- or 96-character structure. Thus, single-byte control characters can be mixed with double-byte graphic characters with no ambiguity.

The rationale for reserving an area for control characters in the right half of an 8-bit set is that communications devices tend to examine only the low-order 7 bits of a character when deciding whether it is a control or graphic character. Placing graphic characters in columns 8 and 9 often triggers unwanted control functions.

During the early decades of computing, when 7-bit communication was the rule, ISO 646 was the predominant method for representing the special characters of each language, and each European country had its own version of ISO 646. But this imposed severe limitations on the user. For example, ISO 646 makes it impossible to mix (say) German text and C-language programming syntax in the same file. The following C program fragment:

```
if (~(a[i] | x)) {
    printf("Grüße aus Köln\n");
}
```

can not be encoded in either ASCII or in German ISO 646, because neither set has all the required characters. If German ISO 646 is used, German special characters are substituted for the braces, brackets, and bars:

```
if (ß(aÄiÜ ö x)) ä
    printf("Grüße aus KölnÖn");
ü
```

whereas if ASCII is chosen, we see the opposite effect:

```
if (~(a[i] | x)) {
    printf("Gr}~e aus K|ln\n");
}
```

Neither result is satisfactory, and matters only deteriorate when German-language program commentary is to be added.

To alleviate these problems, many sites are switching to the ISO 8859 Latin alphabets. However, the ISO 646 versions are still widely used, especially in electronic mail, a predominantly 7-bit medium.

Proprietary Character Sets

Most computers use either US ASCII [1] or IBM EBCDIC [13] as their basic character set. But to support entry, display, printing, and processing of textual data in languages other than English, computer manufacturers soon recognized the need to extend these basic sets to allow representation of the accented Roman letters, non-Roman letters, ideograms, and other symbols used by the world's writing systems.

Some manufacturers provide ISO 646 national versions, but users suffer with their limitations. Besides the sacrifice of characters needed for programming, ISO 646 does not allow mixture of text in different languages, such as Italian, French, and Norwegian, in the same file. So manufacturers such as Digital and IBM began to devise 8-bit international character sets for the European languages, as well as other sets for languages using other writing systems. Most of these 8-bit sets are capable of representing text in several languages, allowing a single product to serve, and work compatibly, over a broader market, for example all of Western Europe.

Prominent among the proprietary sets are IBM's PC code pages [13]. They resemble an ISO Latin Alphabet by having ASCII in the left half, but depart from from the standard structure by using all of columns 8 through 15, including the C1 area, for graphic characters. Thus IBM PC code pages have (at least) 32 more graphic characters than a standard 8-bit character set. Major manufacturers including Apple and NeXT follow the IBM design, but with different character repertoires and encoding. Others, notably Digital, Hewlett Packard, and Data General, observe the standard structure, usually with different repertoires and encoding.

To add to the confusion, we also have IBM's many EBCDIC-based code pages, whose structure does not follow any national or international standard, as well as variations on IBM-like mainframes manufactured in Eastern Europe and the Far East, plus unknown numbers of proprietary sets from other manufacturers.

The Current Situation

Today we are confronted with hundreds of different coded character sets, both standard and proprietary. These sets differ in important ways:

- *Size:* The total code space, 7-bit or 8-bit, single byte or multibyte.
- *Structure:* Standard or nonstandard allocation of control and graphics areas.
- *Repertoire:* The particular selection of characters.
- *Encoding:* The particular code values assigned to each character.

Every application-independent plain-text file is encoded in a particular character set. It is generally not possible to mix character sets within a plain-text file. Furthermore, a text file generally does not contain any indication of its character set. Neither, in general, does the host operating system identify a file's character set, nor indeed, provide any mechanism to do so.

Within most applications and computing environments, a certain character set is simply *assumed*. When a workstation supports multiple character sets, or when data must be communicated between unlike computers, there is no automatic mechanism for software applications to identify a file's character set, and hence no way to automatically display its characters correctly, nor to announce the character set to another computer or application during data transfer.

This problem has grown over the past decade as computers have become increasingly interconnected, and so are used increasingly for communication of text: news, conferencing, file transfer and sharing, and electronic mail. Not only are text character sets likely to be incompatible, but there are no universally accepted methods for translation.

Character Set Translation

Characters, such as the letter A, are represented in the computer, and in telecommunications, by numeric codes. Different computers use different codes for the same character. For example, the letter A is code 65 in ASCII, code 193 in EBCDIC, and code 9025 in JIS X 0208.

The most commonly used translation function is a simple array-indexing operation. Suppose we are translating from character set *A* to character set *B*, and each set has 256 characters, and the characters in each set are represented by 8-bit code values in the range `0..255`. The translation is accomplished by a linear array of 256 8-bit elements called a translation table. The table element at position *i* contains the translation from the character in set *A* whose code is *i* to the corresponding character in set *B*, namely its code in set *B*. For example, the 65th element of an ASCII-to-EBCDIC translation table would be the number 193.

During the translation process, a particular input character, *c_in*, in set *A* is translated to the output character, *c_out*, in set *B* by an array indexing operation as in this example, written in the C programming language:

```
unsigned char a_to_b[256] = { ... };  /* Translation table */
unsigned char c_in, c_out;            /* Input and output characters */

c_out = a_to_b[c_in];                 /* Translation function */
```

where the *c_in* variable is used as a subscript to the *a_to_b* array, and the notation `{ ... }` stands for the initialization of the translation table. In practice, the braces contain the quantities forming the table, in the appropriate order.

In constructing a translation function between any pair of character sets, there are three important and often conflicting goals:

1. *Invertibility* (I): After translating text from *A* to *B*, and then back to *A*, is the result identical to the original?

2. *Readability* (R): After translating text from *A* to *B*, is the result readable?

3. *Consistency* (C): Are translations from *A* to *B* by different applications the same?

In attempting to achieve these goals, we must look at the size, structure, and repertoire of the two character sets. In many cases, the R-versus-I decision is forced upon us, but in others a choice is possible. For example, consider translating between two character sets of the same size: Latin-1 and Latin/Cyrillic. An invertible translation is possible that will not be readable, and a readable translation is possible that is not invertible. In cases like this, the best course is to let the user set the translation goal.

Invertibility

Invertibility is important in cases where the exact contents of the file is important, or when the goal of data transfer is not necessarily final usage. Suppose, for example, you compose a C-language program (in ASCII) on your PC, transfer it to an IBM mainframe (where it is converted to EBCDIC), work on it some more, and then transfer it back to your PC. If the character-set translation from the PC to the mainframe and back is not invertible, you will likely not be able to compile the program again on your PC without syntax errors.

An invertible translation from character set *A* to character set *B* is possible only if *A* is smaller than or the same size as *B*. Similarly, an invertible translation from character set *B* to character set *A* is possible only if *B* is smaller than or the same size as *A*. So it follows that invertibility can be achieved in both directions only if the two character sets are the same size. The following discussion applies only to bidirectional invertibility.

The intersection of the two character sets A and B, written $A \cap B$, is the set of characters, c, that both sets have in common, that is, all the characters are members of ($\in$) both A and B:

$$A \cap B = \{ c \ : \ c \in A, \ c \in B \}$$

The characters in A that are *not* in B can be written as:

$$A \setminus B = \{ c \ : \ c \in A, \ c \notin B \}$$

and the characters in B that are not in A are:

$$B \setminus A = \{ c \ : \ c \in B, \ c \notin A \}$$

To make an invertible translation table, the characters of $A \cap B$ are paired together: the letter "E" in one set to "E" in the other, "È" in one set to "È" in the other, and so on. The characters in $A \setminus B$ are paired 1-to-1 with the characters in $B \setminus A$ according to *some criterion:* readability, consistency, whimsy, or caprice. The exact method for pairing the leftovers is problematic, and frequently a particular pair makes no sense at all, for example "L-with-stroke" with "Vulgar fraction 3/4".

Any 1-to-1 pairing will give an invertible translation, but to achieve the most useful translation it is necessary to examine all the character sets involved. To illustrate, Latin Alphabet 1 lacks the OE digraph character but this character is found in the Digital Multinational character set, the Apple Quickdraw set, the Hewlett Packard Roman8 set, the Data General International set, and the NeXT character set, but at different code points in each. Ideally, the translations for each of these character sets would map OE digraph into the same Latin-1 code point, so that text translated from (say) NeXT to Latin-1 and thence to Data General would keep its OE intact. But this would require an unprecedented degree of cooperation among competing manufacturers.

The construction of invertible translations between private and standard character sets is beyond the scope of the national and international standards organizations, nor should these translations be made arbitrarily by programmers. Translation tables (or algorithms) are most appropriately furnished by the creators or owners of each private character set. This lends the appropriate "official" air and allows all software developers to use the same translations, thus promoting interoperability of diverse applications. In 1990, IBM became one of the few computer manufacturers to take this step when it published invertible tables between ISO 8859-1 and its code pages 500 and 850 in its Character Data Representation Architecture Registry [13].

Other translations, however, are lacking from IBM, for example between its Cyrillic code pages and the ISO Latin/Cyrillic alphabet; similarly for Hebrew, Arabic, Greek, and so on. Official invertible translations of any kind seem to be entirely lacking from most other computer and software makers.

A Simple Rule

In the absence of an official translation, a simple procedure can be used to produce consistent invertible translations across all applications. Let us assume that P is a private nonstandard character set, and that S is a standard character set, and that P and S are the same size, n. Follow these steps to construct the translation table from P to S as an array, *p_to_s*, of n elements:

1. The characters that are common to P and S, $P \cap S$, are mapped together. For each such character in P, whose code value is i, *p_to_s*[i] takes the corresponding character's code value from S.

2. The members of $A \setminus B$ and $B \setminus A$ are paired with each other in *code order*.

This procedure guarantees that *p_to_s* has exactly n unique elements.

Step (1) is not always easy. All too frequently, private character sets are documented only by tables showing the graphic characters, often unclearly (as when working from a fax of third-generation photocopy), with no names or other identifiers assigned to the characters. Even when the material is legible and names are assigned, conventions for graphic representation differ, and so do the names. Thus, some knowledge of languages, writing systems, world and corporate cultures, history, and politics is helpful.

Let's say that character set P consists of four characters, the letters A, B, C, and D, whose code values are 0, 1, 2, and 3, respectively. And set S consists of the letters B, X, A, and Y, also with code values 0, 1, 2, and

3, in that order. The letters A and B are common to both sets. A is represented by code 0 in *P* and by code 2 in *S*, so:

p_to_s[0] = 2

Similarly for the letter B:

p_to_s[1] = 0

Positions 2 and 3 of our translation array remain empty, so we assign them in code order:

p_to_s[2] = 1
p_to_s[3] = 3

and the translation from *P* to *S* is complete. Each element of the *p_to_s* array has a unique value.

To create the reverse translation table from *S* to *P*, *s_to_p*, we could repeat the process in the reverse direction, or, equivalently (and more safely), simply turn the *p_to_s* table "inside out" by sorting it according to its values. Here is a C language program fragment that does the job:

```
for (i = 0; i < n; i++)
  s_to_p[p_to_s[i]] = i;
```

This leaves us with the two translation tables:

Index	*p_to_s*	*s_to_p*
0	2	1
1	0	2
2	1	0
3	3	3

Naturally, such an arbitrary method will please few (hence the foregoing plea for more-sensible *official* invertible translations); in this case C becomes X and vice versa. But *vice-versa* is exactly what is needed for invertibility and consistency. Those characters the two sets have in common are translated readably, and the rest are translated according to the Simple Rule for *consistent* invertibility.

In a more useful application of the Simple Rule, let us construct an invertible mapping between two real-life, 8-bit, single-byte character sets, Data General International (DGI) [7] and ISO 8859-1 [18], between which there is no official invertible mapping. We begin by finding all the characters from DGI that are also in Latin-1 (80 of them) and make the appropriate mappings. We are left with two lists of sixteen unmatched characters. Applying the Simple Rule, the lists are sorted in code order and placed side-by-side to obtain the following correspondence:

160	Undefined	160	No-break space
175	Double dagger	166	Broken bar
179	Trade mark uncircled	173	Soft hyphen
180	Florin sign	175	Macron
183	Less-than-or-equal sign	184	Cedilla
184	Greater-than-or-equal sign	185	Superscript one
186	Grave accent	188	Vulgar fraction one quarter
191	Up arrow	190	Vulgar fraction three quarters
215	Capital OE digraph	208	Capital Icelandic letter Eth
220	Undefined	215	Multiplication sign
221	Uppercase letter Y with diaeresis	221	Capital letter Y with acute accent
222	Undefined	222	Capital Icelandic letter Thorn
223	Undefined	240	Small Icelandic letter eth
247	Small oe digraph	247	Division sign
254	Undefined	253	Small letter y with acute accent
255	Fill character light	254	Small Icelandic letter thorn

So if *P* is DGI and *S* is Latin-1, then *p_to_s*[160] = 160, *p_to_s*[175] = 166, ..., *p_to_s*[255] = 254, and the *P* to *S* mapping is complete. The *s_to_p* array is obtained by exchanging the index and value of each *p_to_s* element, as in the program fragment given above.

Readability

Bidirectional invertibility cannot be achieved when the character sets are different sizes, nor can invertibility be achieved from a larger set to a smaller set. In such cases, readability becomes the only sensible translation goal. Even in cases where invertibility is possible, readability might be preferred for a particular data transfer.

When translating from a larger set, *A*, to a smaller one, *B*, several different characters in *A* can be mapped to a single character in *B*. For example, the following Latin-1 characters:

à á â ã ä å æ

might all be mapped to the letter "a" when translating to ASCII. In the resulting ASCII file, we can't tell where a particular "a" came from, so we can't reconstruct the original Latin-1 text when translating in the reverse direction. But the ASCII file is more intelligible than if we had used some other mapping, such as simply stripping off the high-order bit. Translation by removing diacritics is useful with Roman-based languages, such as French; "pâté" becomes "pate" rather than (say) "pbti". Or German: "Grüße aus Köln" becomes "Gruse aus Koln" instead of "Gr|_e aus Kvln".

In German, the words "Gruse" and "Grüße" have entirely different meanings (we don't want to say "soot" when we mean "greetings"). We can do better. European languages like German, Swedish, Norwegian, Danish, Icelandic, and Dutch have rules for converting accented or other special characters into unadorned ABC's. For example, any German vowel with an umlaut (diaeresis) can be written without the umlaut and followed by the letter "e". These rules are specific to each language. So while we can write the German word "Köln" as "Koeln", we cannot write the English word "coöperation" as "cooepoeration".

Such language rules can not be applied blindly in reverse. For example, if "oe" were translated back to "ö", then "Kommandoebene" would become "Kommandöbene" (not a German word), and `AUTOEXEC.BAT` would become `AUTÖXEC.BAT` (a PC file that you don't want to rename!).

Construction of a readable translation between two entirely different alphabets, such as Cyrillic and Roman, is called transliteration. The specific transliteration rules must take into account not only the alphabets themselves, but also what languages they represent. For example, the surname of a former leader of the former USSR, `Kru7ev`, is transliterated into Roman letters as ‘‘Khrushchev" in English, but into "Khruschtschew" in German.

Newspapers and magazines, libraries, immigrant bureaus, and other organizations have their own standard procedures for transliterating "foreign" writing systems. Not just in "ASCII-speaking" lands, but everywhere: Russian names are written in Arabic newspapers, Hebrew names in Greek journals, English names on Chinese passports, Korean publications in Vietnamese library catalogs. But these standards are not widely known. When a standard can be found, use it. If not, look harder.

Character-Set Translation in the Kermit File Transfer Protocol

The Kermit File Transfer Protocol was developed at Columbia University to allow the transfer of both text and binary files among all types of personal computers, minicomputers, and mainframes, in both the 7-bit and 8-bit communication environments. Kermit is a layered, point-to-point, transport-independent, error-correcting packet protocol described in detail elsewhere [5].

Transfer of text files between unlike computers requires conversion of both record format and character set at the presentation layer. For example, a document composed under the UNIX operating system using the ASCII character set with lines separated by imbedded Linefeed characters, upon transfer to an IBM mainframe, is converted to EBCDIC encoding and a mainframe-specific variable- or fixed-length record format. Kermit accomplishes this conversion with another Simple Rule: during file transfer, the character set used for text files is ASCII, and the record format is stream, with records (lines) delimited by Carriage Return and Linefeed.

Thus, it is the responsibility of each Kermit program to convert between the text character sets and record formats of its own computer and the standard Kermit format. This means that no Kermit program needs to know the specific codes and formats of any kind of computer except its own, and it forms the basis of Kermit's strategy for converting between different character sets. This idea is known as a "common intermediate representation," and it lies at the heart of any presentation-layer protocol [26].

By the mid 1980s, Kermit had become a de facto standard for file transfer. Kermit software programs had been written for almost every kind of computer in existence. But the Kermit protocol lacked a formal and consistent means for exchanging text that contained non-Roman or accented Roman characters. Files could be transferred, but the results would be gibberish unless the receiving computer supported the same character set as the sender.

At first, this problem was remedied by pre- or postprocessing. But this approach places an unreasonable burden on the user. Not only must extra steps be taken, but a suitable translation utility must be found for every pair of character sets. More subtly, translation utilities (for example, between an IBM code page and a Macintosh character set) are constructed in an ad-hoc manner, with no guarantee of consistency from one utility to another.

The problem is compounded by the rapid proliferation of proprietary, national, and international standard character sets. By the late 1980s, there were many encodings for each major writing system, a problem pointed out by attendees at international conference sessions on Kermit in Europe and Japan [10]. A consistent approach to character-set translation had become an urgent matter.

Basic Design Principles

How can we enable meaningful exchange of text between any two computers? The obvious approach is to require each data-transfer application to understand every character set in existence. This works adequately when the number of sets is small and stable, but quickly becomes unwieldy and unmanageable as the number increases. If the number of character sets is *n*, the number of translations is:

$$\binom{n}{2} = \frac{n\,!}{2!\times(n-2)\,!} = n\times(n-1) \qquad \textbf{(1)}$$

If we have two character sets, *A* and *B*, we need two translations, one from *A* to *B* and one from *B* to *A*. If we have three sets—*A*, *B*, and *C*—we need $3\times2 = 6$ translations: *AB*, *BA*, *AC*, *CA*, *BC*, and *CB*. And so on.

Now consider that in 1990, IBM alone listed 276 different coded character-set identifiers in its registry [13]. If we wanted translations between every pair of IBM character sets, there would be 75,900 of them! Add in all the other sets from all the other companies to appreciate the magnitude of the problem.

By using a standard intermediate representation for each type of character set (Roman, Cyrillic, Hebrew, Japanese, etc), we eliminate the need for any particular computer to know about the character sets used by any other kind of computer. Kermit's common intermediate character set, previously always ASCII, is now allowed to be any of a small number of character sets. The set used during a particular file transfer is called the transfer character set (TCS).

The character set of the file that is being sent or received is called the file character set (FCS). The sender translates from its local codes (the FCS) to the standard ones (the TCS), and the receiver translates from TCS codes to its own FCS, as shown in Figure 2.

For a particular file and transfer character set combination, a Kermit program has one translation function for sending files and another for receiving them. Theoretically, all combinations of file and transfer character set are allowed. Thus the number of translation functions, *f*, is given by:

$$f = tcs \times fcs \times 2 \qquad \textbf{(2)}$$

That is, one function in each direction, for each combination of TCS and FCS. While this number is significantly lower than the number of pairs of all character sets (Equation 1), we still want to reduce it to conserve computer memory and cut down on user confusion.

The Transfer Character Set

Generally, we want to support all the file character sets used on a particular computer, so the way to keep the total number of translation functions small is to minimize the number of transfer character sets that can handle the given selection of file character sets. This is done, in part, by restricting the set of *possible* transfer character sets to a small number according to the following rules:

```
     COMPUTER A                                    COMPUTER B
+------------------+                          +------------------+
| +-------------+  |                          |  +-------------+ |
| | Translation |  |      Transfer            |  | Translation | |
| | Function:   |--------------------------->|  Function:   | |
| | FCS to TCS  |  |    Character Set         |  | TCS to FCS  | |
| +-------------+  |                          |  +-------------+ |
|       ^          |                          |        |         |
|       |          |                          |        v         |
|  Kermit Program  |                          |  Kermit Program  |
|      SEND        |                          |     RECEIVE      |
+------------------+                          +------------------+
        ^                                              |
        |                                              v
+------------------+                          +------------------+
|  Local File      |                          |  Local File      |
|  Character Set A |                          |  Character Set B |
+------------------+                          +------------------+
```

Figure 2: File Transfer Character-Set Translation

1. The transfer character set must be a national or international standard character set registered with the ISO, or a combination of such sets. This means that its structure is consistent with other standard character sets and that it has a unique identifier. Furthermore, it means that the character set is well known, its specification is readily available, and the characters have names.

2. US ASCII [1] is included for compatibility with the original Kermit protocol and with unextended Kermit programs.

3. The ISO 8859 Latin Alphabets [18] are included.

4. The ISO-registered Chinese [4], Japanese [22, 23, 24], and Korean [25] sets are included. These are usually used in conjunction with one or more single-byte sets that provide control characters and single-width ASCII or ISO 646 graphics.

5. Additional sets, such as (for example) Vietnamese VSCII [8, 28], can be included if they are registered with the ISO, as long as they are not proper subsets of any of those already included.

6. All else being equal, a simple and compact representation is preferred.

The national versions of ISO 646 [14] (other than US ASCII) are not included because of Rule 5; these sets are covered adequately by the ISO Latin alphabets. CCITT (ITU-T) T.61 [2], which represents accented characters exclusively by composition, is not included for reasons 1, 5, and 6.

Table 1 lists the transfer character sets presently allowed by the Kermit protocol. The Kermit Name allows uniform reference to these sets by Kermit software users. The requirement for ISO registration provides for unique and incontestible identifiers for Kermit's transfer character sets. The Kermit Designator is the means by which the sending Kermit program informs the receiver of the transfer character set, a key part of the presentation protocol. As noted earlier, however, ISO standards do not provide a single designator for a complete character set, but rather separate designators for its pieces. Thus Latin-1 is designated as "I6/100", meaning that the left half (G0) is ASCII and the right half (G1) is "the Right-hand Part of Latin Alphabet 1." The C0 and C1 control regions are not explicitly designated. The C0 region is assumed to be the normal ASCII and ISO 646 control set, with format effectors used to delimit records and so on. The C1 set is assumed to be ISO 6429 to allow the use of character-set shifting functions such as SS2 and SS3 [15], for example in Japanese EUC.

In the Kermit Designator, the initial letter "I" indicates ISO registration numbers for character sets, leaving open the possibility for other registration authorities. Japanese EUC (Extended UNIX Code) is a special case, having three parts, chosen in preference to JIS X 0208 alone to allow the commonly used mixture of

Table 1: Kermit Transfer Character Sets

Kermit Name	*Standard*	*ISO Registration Number*	*Kermit Designator*	*Languages*
ASCII	ANSI X3.4	6	*(none)*	English, Latin
LATIN1	ISO 8859-1	100	I6/100	Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, and Swedish.
LATIN2	ISO 8859-2	101	I6/101	Albanian, Czech, English, German, Hungarian, Polish, Romanian, Croatian, Slovak, and Slovene.
LATIN3	ISO 8859-3	109	I6/109	Afrikaans, Catalan, Dutch, English, Esperanto, French, Galician, German, Italian, Maltese, Spanish, and Turkish.
LATIN4	ISO 8859-4	110	I6/110	Danish, English, Estonian, Finnish, German, Greenlandic, Lappish (Sami), Latvian, Lithuanian, Norwegian, and Swedish.
LATIN5	ISO 8859-9	148	I6/148	Danish, Dutch, English, Faeroese, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, and Turkish.
CYRILLIC	ISO 8859-5	144	I6/144	Bulgarian, Byelorussian, English, Macedonian, Russian, Serbocroatian (Serbian), and Ukrainian
ARABIC	ISO 8859-6	127	I6/127	Arabic
GREEK	ISO 8859-7	126	I6/126	Greek
HEBREW	ISO 8859-8	138	I6/138	Hebrew
KATAKANA	JIS X 0201	14, 13	I14/13	Japanese (Roman and Katakana)
JAPANESE-EUC	JIS X 0201, JIS X 0208	14, 13 87	I14/87/13	Japanese (Roman, Katakana, Hiragana, and Kanji), English, Greek, Russian
CHINESE	CS GB 2312-80	58	I55/58	Chinese (Roman, Phonetic, and Hanzi), Japanese (Roman, Katakana, Hiragana), English, Greek, Russian
KOREAN	KS C 5601	149	I6/149	Korean (Hangul, Hanja), Japanese (Roman, Katakana, Hiragana), Greek, Russian, English, and others
VIETNAMESE	TCVN 5712	180	I6/180	Vietnamese

single-width and double-width characters. The registration numbers are listed in G0/G1/G2 order, so SS2 is required to shift between Kanji (ISO 87) and Katakana (ISO 13) in accordance with ISO 2022.

This notation is used in preference to, say, the name of the standard itself (for example ‘‘ISO8859-1’’) because the same character set can be defined by more than one standard (for example, Latin-1 and ECMA 94), and one standard can specify more than one character set (e.g. ISO 646).

Using a standard international character set as the TCS, it is possible to transfer text written in a language other than English between unlike computers, and it is usually also possible to transfer text containing a mixture of languages. For example, text in Latin Alphabet 1 might contain a mixture of Italian, Norwegian, French, German, English, and Icelandic.

A particular Kermit program need not incorporate all the defined transfer character sets. In many cases, a single 8-bit set will suffice, such as LATIN1 for Western Europe, LATIN2 for Eastern European languages with Roman-based writing systems, CYRILLIC for Russia, and so on. Thus Equation 2 generally results in a comfortably small number. For example, an IBM PC that supports five Roman-alphabet code pages for Western European languages plus a Cyrillic code page can be used with two transfer character sets, LATIN1 and CYRILLIC for a total of 24 translation functions.

When a language is representable in more than one set from Table 1, as are English, German, Finnish, Turkish, Greek, Russian, etc., the character set highest on the list that adequately represents the language should be used. For example, ASCII should be used for English. Within the ISO 8859 family, lower-numbered sets that contain all the characters of interest are preferred to higher-numbered sets containing the same characters.

This guideline maximizes the chance that any two particular Kermit programs will interoperate. For example, LATIN1 would be chosen for French, German, Italian, Spanish, Danish, Dutch, Swedish, etc; LATIN3 for Turkish; JAPANESE-EUC for Japanese text that includes Kanji characters, KATAKANA for Japanese text that includes only Roman and Katakana characters, and so on.

If a file containing a mixture of languages, say English, Finnish, and Latvian, must be transferred, the user must find a transfer character set that can adequately represent all three languages, in this case Latin Alphabet 4. For a mixture of Norwegian and Turkish, Latin-5 must be used, and so on.

The user can employ this flexibility to achieve useful effects. For example, since there is no requirement that a Cyrillic file be transferred using a Cyrillic transfer character set, the user can transliterate between Cyrillic and Roman characters as part of the file transfer process.

The Translation Function

A typical Kermit program contains an $m \times n \times 2$ matrix of translation functions, where m is the number of supported file character sets, n is the number of supported transfer character sets (including the *transparent* set, which indicates that no translation is to be done), and there are two tables for each combination, one for sending and one for receiving. The translation function is selected when the user identifies the file and transfer character sets and then sends or receives a file.

The normal behavior of a particular function can be altered in several ways. The user can override its default translation goal, and when the goal is readability (as it must be when translating from a larger to a smaller set), language-specific rules can be invoked.

The function itself can work by any combination of algorithm, translation table, exception list, and shameless tricks. For example, an invertible translation between IBM Code Page 437 and Latin Alphabet 1 would be a simple indexing operation into a table, but a translation from Japanese EUC to the PC "Shift JIS" code is normally accomplished by a tableless algorithm. Translation from Latin-1 to ASCII with German language rules could be done with a combination of table accesses and exception lists. To accomplish the desired translation, each Kermit program needs to know:

- The local file character set.
- The transfer character set to be used.
- The translation goal, invertibility or readability.
- For readable translations, optionally, the language and a corresponding set of language-specific rules.

In most situations, some or all of these are implicit, and no particular efforts are required. To illustrate, suppose that you have an IBM PC on your desk, and the PC is connected to a Hewlett Packard (HP) timesharing computer. The PC uses IBM code page 850 and the timesharing computer uses the HP Roman8 Set. In that case, your PC's file character set is always CP850, the timesharing computer's file character set is always HP Roman8, and the transfer character set is always Latin-1. These items can be set in your Kermit profiles, and the appropriate translations will always occur automatically.

On the other hand, suppose you must occasionally write some text in German and send it from your PC to another computer that supports only ASCII. In this case you would override your Kermit profiles by specifying a transfer character set of ASCII, which automatically activates the readability goal, and you might also choose to elect language-specific rules for German.

Examples

Let's look at a few of many possible translation scenarios. Each one presents its own set of problems requiring decisions by the creator of the translation function, or by the user.

1. From a 7-bit set to a different 7-bit set, e.g. from the Spanish version of ISO 646 to ASCII (or vice versa). The two sets do not contain the same characters. Here we must choose between readability (R) and invertibility (I). To achieve readability in the Spanish-to-ASCII direction, we strip diacritical marks (n-tilde becomes simply n, and so on). To achieve invertibility, we make no translation at all.

2. From a 7-bit set to an 8-bit set. The 7-bit sets are usually ASCII or an ISO 646 national version. Often, all the characters from the 7-bit set are also present in the 8-bit set, and there is no R-versus-I conflict. For example: ASCII (and most ISO 646 national variants) to Latin-1—here we satisfy both R and I. In other cases we must choose between R and I. For example: the ISO 646 Italian national variant to ISO Latin / Arabic: here we either remove the accents for readability, or map the accented characters into right-half characters for invertibility.

3. From an 8-bit set to another 8-bit set. A common case is converting between one of the corporate "extended ASCII" sets (Digital, IBM, HP, Apple, NeXT, Data General) and ISO Latin-1. The two sets share a large percentage of common characters. How do we handle the characters that differ? Again, we must choose between R and I. To complicate matters, the IBM, Apple, and NeXT sets use the forbidden C1 control-character area for graphic characters. To create an invertible translation in the absence of an official corporate standard, we use the Simple Rule.

4. From an 8-bit set to a 7-bit set. For example, from Latin-1 to ASCII or to an ISO 646 national set. Here we are forced to accept a large amount of information loss. We cannot possibly achieve invertibility, so we aim for maximum readability, for example by removing diacritics or invoking language-specific rules.

5. From a single-byte character set to a multibyte character set. Most multibyte character sets include ASCII and sometimes several other alphabets (such as Greek and Cyrillic). Here we translate each character into its equivalent, if it has one. When it doesn't, we must choose between R and I. For example, "Ö" is not found in JIS X 0208 so it can be mapped to "O" for readability, or some unique value (preferably one unassigned in JIS X 0208) for invertibility.

6. From a multibyte set to a single-byte set, for example Japanese JIS X 0208 into Latin-1 (or Latin/Cyrillic, Latin/Greek, or even ASCII). An invertible translation is clearly impossible. A readable translation would require rendering Kanji ideograms phonetically or translating them into an entirely different language, clearly beyond the scope of a character-set conversion scheme.

7. From one national multibyte set to another. These sets are for Chinese, Japanese, and Korean, and have a very large number of characters—ideograms, ASCII graphics, Greek, and Cyrillic characters—in common. They also have large blocks of unassigned character positions, so the characters they do not share in common (such as the Chinese phonetic symbols that are absent from the standard Japanese set) can be assigned to these areas to preserve invertibility.

No two programmers are likely make the same decisions and this will lead to inconsistent translations (unless the Simple Rule is followed). This emphasizes the need for officially published translations between the private and standard sets. And as this list suggests, we also need translations between some of the standard sets themselves, for example Chinese and Korean. This need is addressed to some extent by the Unicode books [27], which include the mappings from various character sets to the Unicode set.

Performance

Character-set translation in itself does not affect the performance of the Kermit file transfer protocol to any significant degree. The introduction of per-character translation introduces an extra table access or function call but the extra work is usually minimal. In general, the bottlenecks are elsewhere.

One of the strong points of the Kermit protocol is its ability to transfer 8-bit data in the 7-bit communication environment. This is done using a single shift, or prefixing, technique in which each 8-bit character is stripped of its 8th bit and then prefixed by a special shift-indicating character. This results in negligible overhead for English and Western European text (such as French, German, Italian).

But for text in ‘‘right-handed’’ languages like Russian, Greek, Hebrew, and Arabic, where text characters come predominantly from the right half of the character set, single shifts can result in up to 80% overhead. The situation is even worse for Japanese EUC, in which all Kanji bytes have their 8th bits set to 1, resulting in transmission overhead of 100% for pure Kanji text on a 7-bit connection.

Because 7-bit communication is still prevalent, Kermit’s support for Greek, Cyrillic, Hebrew, Arabic, and Japanese text file transfer calls for a more efficient technique. This was accomplished by adding a locking shift mechanism to the Kermit protocol, allowing sequences of 8-bit characters to be transmitted in 7-bit form with shifting overhead applying to entire 8-bit sequences, rather than to each 8-bit character. Isolated 8-bit characters can still be transmitted using single shifts. These methods are very similar to those of ISO 2022, but without the risk of ‘‘loss-of-state’’ due to corruption or loss of the shift characters, and with the addition of the ‘‘single-shift-1’’ mechanism lacking from ISO 2022. A combination of single and locking shifts can achieve maximum efficiency by using a lookahead technique. A detailed specification is given elsewhere [12].

The addition of locking shifts to the Kermit protocol increases the transfer efficiency on 7-bit connections for typical Cyrillic text by about 50% and for typical Kanji text by more than 90%, bringing these transfers to within the efficiency range of normal 7-bit ASCII transfers on the same connections.

Conclusions and Recommendations

File transfer character-set translation is an optional feature for Kermit programs, and is designed to interoperate (with, of course, no claim to correct translation) with Kermit programs that do not support it. As of this writing, translation of (at least) Roman, Cyrillic, Hebrew, and Japanese text is supported by MS-DOS Kermit for the IBM PC [9], IBM mainframe Kermit for VM, MVS, and CICS [3], and C-Kermit for UNIX, VMS, OS/2, and other operating systems [6]. Three basic commands were added to these programs to select the file and transfer character sets and any desired language-specific translation rules. Locking shifts are used automatically in the 7-bit communication environment to prevent Kermit from discriminating against ‘‘right-handed’’ character sets. Kermit programs equipped with the new translation features have become popular in Europe, Latin America, the former Soviet Union, and Japan. Work is in progress to add further translation capabilities for other parts of the world.

Space has not permitted discussion of the details of the Kermit protocol, the forms of the commands, translation negotiation and refusal mechanisms, unilateral and local translation features, automatic matching of file and transfer character sets, character sets in terminal emulation, and numerous other issues. These will be covered in a future edition of reference [5].

In the discussions that resulted in the character-set translation extension to Kermit, the most fundamental lesson we have learned is that if existing standards can solve a particular problem, they should be used instead of inventing new techniques to solve the same problem. Applying this lesson to Kermit file transfer results in the following conclusions:

1. Only ISO-registered standard character sets should be used for interchange. This eliminates the need for any computer to support any character sets except its own and the corresponding well-known standards.

2. The sender should identify the transfer character set to the receiver using a standard notation such as its ISO registration number. This eliminates the need for setting up separate and redundant registration authoritities for character-set identifiers.

3. Translations should be invertible, readable, and consistent. When all three of these goals cannot be achieved by a single translation, the user should be able to choose the translation goal.

These principles can be applied to any form of textual data interchange, including electronic mail, network file systems, terminal emulation, virtual terminal service, distributed databases, remote procedure calls, cutting and pasting among object-oriented applications, and so on.

The translation process itself, however, remains ill-defined. It is hoped that the industry and the standards organizations will take the following steps:

1. Standard character sets should be used in preference to private character sets.
2. Owners of private character sets should publish official invertible translations to ISO-registered standard sets. In the absence of official translations, a simple procedure such as the one presented in this paper should be used to achieve consistent invertible translations across all applications.
3. Standards organizations are encouraged to consider publishing translations between different standard character sets, such as the Japanese, Chinese, and Korean sets, as well as readable transliterations among different alphabetic writing systems, such as Roman, Greek, Hebrew, Arabic, and Cyrillic.
4. Operating system designers should consider tagging plain-text files with character-set identifiers, like the Kermit tags listed in Table 1, to allow applications software to determine a file's character set automatically. When standard character sets are used, their tags should be consistent across different operating systems.

Ten or twenty years from now, perhaps all the computers, as well as all the display, entry, printing, and telcommunication devices of the world will use one universal character set, and the issues discussed in this paper will be irrelevant. On the other hand, perhaps the accumulated and ever-growing installed base of existing hardware, software, and electronic information will prove too massive for conversion and the universal character set will be just one more character set on the list.

Acknowledgements

My deepest thanks to Christine M. Gianone of Columbia University for inspiring the work and ideas described in this paper, and for her key contributions thereto. Special thanks also to the others who played prominent roles in the design and development of Kermit's character-set translation capabilities: Joe R. Doupnik of Utah State University, John Chandler of the Harvard/Smithsonian Center for Astrophysics, Hirofumi Fujii of the Japan National Laboratory of High Energy Physics in Tokyo, John Klensin of the United Nations University, André Pirard of the University of Liège in Belgium, Johan van Wingen of the Netherlands and numerous ISO committees, Gisbert W. Selke of the Wissenschaftliches Institut der Ortskrankenkassen in Bonn, Germany, and Konstantin Vinogradov of the International Centre for Scientific and Technical Information (ICSTI) in Moscow. Grateful acknowledgements also to Juri Gornostaev and A. Butrimenko of ICSTI for hosting the First International Kermit Conference in Moscow [11] in Spring 1989, where the ideas in this paper received their first public hearing. Thanks also to the many participants in the ISO8859, UNICODE, and ISO10646 network discussion groups for valuable information and insights.

About the Author

FRANK DA CRUZ is Manager of Communication Software Development at Columbia University, author of the book *Kermit, A File Transfer Protocol*, co-author (with Christine M. Gianone) of the book *Using C-Kermit*, leader of the team that developed the Kermit protocol, and principal author of several Kermit software programs including C-Kermit for UNIX, VMS, and OS/2. Present address: Columbia University, 612 West 115th Street, New York, NY 10025, USA; Email: fdc@columbia.edu.

References

1. *ANSI X3.4-1986, Code for Information Interchange.* American National Standards Institute, 1986. The ASCII specification; the US version of ISO 646.

2. *CCITT Recommendation T.61, Character Repertoire and Coded Character Sets for the International Teletex Service.* CCITT, Geneva, 1980 (amended 1984).

3. Chandler, John. *IBM System/370 Kermit User's Guide.* Columbia University Academic Information Systems, 1993. Available in separate versions for VM/CMS, MVS/TSO, and CICS.

4. *Chinese Standard GB 2312-80, Coded Chinese Graphic Character Set for Information Interchange.* China Association for Standardization, Beijing, 1980.

5. da Cruz, Frank. *Kermit, A File Transfer Protocol.* Digital Press, Bedford, MA, 1987.

6. da Cruz, Frank and Christine Gianone. *Using C-Kermit.* Digital Press, Burlington, MA, 1993. EY-J896E-DP, German edition available Fall 1993.

7. Data General. *Programming the Display Terminal: Models D217, D413, and D463.* Data General, Westboro, MA, 1991. 014-002111-00.

8. Do, James, Ngô Thanh Nhàn, Hoàng Nguyên. "A proposal for Vietnamese character encoding standards in a unified text processing framework". *Computer Standards & Interfaces 14* (1992).

9. Gianone, Christine M. *Using MS-DOS Kermit.* Digital Press, Burlington, MA, 1992. EY-H893E-DP, Also available in French and German editions.

10. Gianone, Christine M. "Have Kermit, Will Travel". *Kermit News 3*, 1 (June 1988).

11. Gianone, Christine M. "Mission to Moscow". *Kermit News* , 4 (June 1990).

12. Gianone, Christine M. and Frank da Cruz. A Locking Shift Mechanism for the Kermit File Transfer Protocol. Columbia University, 1991.

13. *IBM Character Data Representation Architecture, Level 1 Registry.* IBM Canada Ltd., National Language Technical Centre, Ontario, 1990. SC09-1391-00.

14. *ISO Standard 646, 7-Bit Coded Character Set for Information Processing Interchange.* Second edition, International Organization for Standardization, 1983. Also available as ECMA-6, and similar to CCITT T.50.

15. *ISO International Standard 2022, Information processing — ISO 7-bit and 8-bit coded character sets — Code extension techniques.* Third edition, International Organization for Standardization, 1986. Also available as ECMA-35.

16. *ISO International Standard ISO 2375, Information processing — Procedure for Registration of Escape Sequences.* International Organization for Standardization, 1985.

17. *ISO International Standard 4873, Information processing — ISO 8-bit code for information interchange — Structure and rules for implementation.* Second edition, International Organization for Standardization, 1986. Also available as ECMA-43.

18. *ISO International Standard 8859 Parts 1 through 9, Information Processing—8-Bit Single-Byte Coded Graphic Character Sets.* International Organization for Standardization, 1987–. ISO 8859-1 through -4 are the Latin Alphabets 1 through 4, also available as ECMA-94. ISO 8859-5 is the Latin/Cyrillic Alphabet (ECMA 113).

19. *ISO/IEC 10646-1, International Standard 10646, Information Technology—Univesral Multiple-Octet Coded Character Set (UCS).* ISO/IEC JTC1, 1993.

20. *ISO International Standard 6429, Information processing — C1 Control Character Set of ISO 6429.* International Organization for Standardization, 1983.

21. *ISO International Register of Coded Characters to Be Used with Escape Sequences.* European Computer Manufacturers Association (ECMA), 1990, updated periodically.

22. *JIS X 0201, The Japanese Katakana and Roman Set of Characters.* Japan Industrial Standards Committee, 1969.

23. *JIS X 0208, The Japanese Graphic Character Set for Information Interchange.* Japan Industrial Standards Committee, 1983.

24. *JIS X 0212, Supplementary Japanese Graphic Character Set for Information Interchange.* Japan National Committee on ISO/IEC JTC1/SC2, 1991.

25. *Korean Standard KS C 5601-1987, Korean Graphic Character Set for Information Interchange.* Korea Bureau of Standards, 1987.

26. Padlipsky, M. A. *The Elements of Networking Style.* Prentice Hall, 1985.

27. The Unicode Consortium. *The Unicode Standard, Worldwide Character Encoding, Version 1.0.* Addison-Wesley Publishing Company, Volume 1, 1991; Volume 2, 1992.

28. Viet Nam General Department for Standardization. *Vietnamese National Standard TCVN 5712, 8-bit Vietnamese Standard Code for Information Interchange (VSCII).* Viet Nam State Committee for Sciences, 1993.