Character encodings

Andreas Fester

June 26, 2005

Abstract

This article is about the concepts of character encoding. Earlier there were only a few encodings like EBCDIC and ASCII, and everything was quite simple. Today, there are a lot of terms like UNICODE, Codepage, UTF8, UTF16 and the like. The goal of this article is to explain the concepts behind these terms and how these concepts are supported in programming languages, especially Java, C and C++.

1 Introduction

Character encodings are necessary because the computer hardware can only handle (and only knows) about numbers. The first computers could handle eight bit numbers (lets take aside the very first microprocessors which could only handle 4 bits), which is a range from 0 to 255. Larger numbers must be handled by combining two or more eight bit values. Later, computers were enhanced so that they could handle 16-, 32- and meanwhile 64 bit numbers without having to combine smaller units. But - its still all numbers, i.e. a sequence of 0 and 1 bits. However, it soon became clear that interfacing with a computer system through sequences of 0 and 1 is very cumbersome and only possible for very simple communication. Usually an end user wants to read text like "Name:" and enter words like "Bill", so it was necessary to define some conventions how numbers correspond to letters. These conventions are generally called *character encodings*.

2 Single byte encoding

The traditional way of assigning a code to a character is to represent each character as a byte or a part of a byte (for example six or seven bits).

2.1 EBCDIC

EBCDIC is the code used by IBM to encode characters for their mainframe systems. It is the abbreviation for *Extended Binary Coded Decimal Interchange Code*. As described by this name, the code is based on BCD, the *Binary Coded Decimal* encoding which is used to represent digits in four bits. With BCD, a single byte can represent up to two digits. The advantage of this code is that converting a number into the separate digits (for example to display them) is very easy. The disadvantage is that not all bit combinations of a byte are used. Only those bit combinations which represent digits from 0 to 9 (i.e. 0000 to 1001) are used, the bit combinations from 1010 to 1111 are not. Similarly, EBCDIC does not use all possible bit combinations to represent characters.

2.2 Seven bit ASCII code

ASCII (*American Standard Code for Information Interchange*) is one of the first character encodings and probably the one which is widest spread. It assigns one letter to each of the numbers in the range from 0 to 127. These are only 7 bits, because in the early days the eight' bit was usually used as parity bit to detect transmission failures.

The ASCII code can be divided in several subsections:

- Values 0 to 31 contain special control characters, such as the backspace or the bell character.
- Values 32 to 47 contain characters like the percent sign, brackets and the ampersand.
- Values 48 to 57 contain the numbers 0 to 9.
- Values 58 to 64 again contain some special characters like the at sign and the colon
- Values 65 to 90 contain the capital letters from A to Z
- Values 91 to 96 again contain some special characters like the circumflex
- Values 97 to 122 contain the non capital letters a to z
- Values 123 to 127 contain some more special characters like the tilde.

2.3 ANSI character encoding

ASCII was standardized as ANSI standard X3.4.

2.4 Extended ASCII code

With computers getting more robust, the need for using the eight' bit as parity bit was not necessary anymore, especially for local usage where no remote transmission is necessary. Therefore, the remaining 128 characters which have not been used in the original ASCII encoding have been assigned more special characters, such as german umlauts, french letters with accents and also some mathematical symbols.

2.5 Codepages

Even with the extended ASCII encoding, a lot of characters were still missing and could not be displayed, especially from languages which use non-latin alphabets such as cyrillic. To make it possible to also display these characters, the encoding of the 128 upper characters of the extended ASCII code was made more flexible: the same number now was mapped to more than one character. Since this mapping is now not unique anymore, it was necessary to specify which encoding to use for a given text. If the wrong encoding was used to display the text some characters were simply displayed as the wrong character. Microsoft implemented this concept in its MS-DOS and MS-Windows operating systems by so called *code pages*. Each code page defines one specific mapping for the numbers from 128 to 255. For example, in code page 437, the value 228 is mapped to the greek sigma character, but in code page 850 it is mapped to the character "o" with a tilde above it.

This approach has the advantage that eight bits were still sufficient and that existing software (especially regarding string handling) did not need to be modified. The operating system and the graphics adapter took care of the proper rendering of the characters, depending on the code page which was configured.

However, this approach has also two large drawbacks. First, the code page must be properly and consistently configured. For example, if the word processor uses code page 437, but the printer uses code page 850, all characters which are different between these code pages are printed differently than they are displayed on the screen. Second, only characters from one code page can be displayed at the same time. Displaying a text containing letters from two different code pages (for example a german text with umlauts, containing some french or czchech cites) at the same time is simply not possible.

2.6 ISO 8859

Besides Microsofts code pages there are a number of standardized character encodings available. Especially the ISO 8859 character encoding is quite wide spread. It contains the mappings shown in table Table 1.

Table 1: ISO 8859 character encodings

ISO character set	Characters contained within
ISO 8859-1	Latin-1 (western europe)
ISO 8859-2	Latin-2 (eastern europe)
ISO 8859-3	Latin-3 (southern europe and esperanto)
ISO 8859-4	Latin-4 (Baltic)
ISO 8859-5	Cyrillic
ISO 8859-6	Arabique
ISO 8859-7	Greek
ISO 8859-8	Hebrew
ISO 8859-9	Latin-5 (Tourquise)
ISO 8859-10	Latin-6 (Nordique)
ISO 8859-11	Thai
ISO 8859-12	Keltique (not finalized!)
ISO 8859-13	Latin-7 (Baltique)
ISO 8859-14	Latin-8 (Keltique)
ISO 8859-15	Latin-9 (western europe including euro sign)
ISO 8859-16	Latin-10 (south eastern europe including euro sign)

3 Multi Byte encoding

So far, each single character was encoded in one byte. This was very easy and straightforward, even for the earlier computer systems. But the problem is that a specific encoding must be used to properly display the characters, and displaying characters from different encodings at the same time is not possible. Since memory and processing time cost has decreased significantly, a new approach could be implemented: the usage of more than one byte for a single character, so that more than 255 characters can be displayed at the same time.

3.1 Internal and external representation

Even though memory costs have decreased, it still makes sense to avoid unnecessary memory consumption. Especially when data is transferred through a network it is still important to reduce the amount of data as much as possible to reduce the time for the data transfer. For example, when using a multi byte concept which uses 2 bytes for each character to make it possible to encode as much as 65536 characters, the amount of data is doubled compared to the old ASCII encoding. Therefore, text is usually represented differently inside a program than it is represented outside a program (on disk, or when transmitted through a network).

The internal representation is usually implemented in a way to make handling the text in a specific programming language as easy as possible. For example, determining the length of a string is much simpler if each character has the same length of lets say two bytes. On the other hand, when storing the text to disk, it probably contains a lot of zero values which can be used to "compress" the text and let it consume less space on the disk. This is called the *external representation*.

4 UNICODE

Similar to the single byte ASCII encoding from the early days it is also necessary to define a unique mapping from numbers to characters in multi byte environments. The probably widest known mapping today is UNICODE. It includes all characters from the legacy ISO or code page encodings, and many more; and it allows to display all these characters at the same time because it uses more than one byte for the encoding. For example, refering to the sample above, the greek sigma character and the "o" with a tilde above which both map to value 228 in two different code pages will now map to different values in UNICODE.

5 ISO 10646

ISO 10646 defines the Universal Character Set (UCS). In practice, this standard is identical to UNICODE.

6 UTF

UTF is the abbreviation for "Unicode Transformation Format". It defines methods to store a Unicode character into a sequence of bytes, and is therefore an *external representation* for UNICODE characters.

6.1 UTF-8

In UTF-8, the smallest unit is the byte. To be more specific, each character from the ancient ASCII character set is encoded identical to how it was encoded in the ASCII character set. If the highest bit is set to "1" however, the encoding becomes more complicated, as it uses more than one byte. The first byte always starts with a sequence of "1" bits followed by a terminating "0" bit:

- 0xxxxxxx => Ancient ASCII character set, 7 bits can be used for the character.
- 110xxxxx 10xxxxxx => two bytes are used to encode the character, of which 11 bits can be used for the actual character encoding.
- 1110xxxx 10xxxxxx 10xxxxxx => three bytes are used to encode the character, of which 16 bits can be used for the actual encoding.
- 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx=> four bytes are used to encode the character, of which 21 bits can be used for the actual encoding.

The above list shows the following implications:

- The number of "1" bits before the first "0" bit in the first byte of each character determines the number of bytes used to encode the whole character.
- Each "follow up" byte starts with "10". Each "startup" byte starts with "110", only the single byte ASCII range starts with a single "0".

6.2 UTF-16

UTF-16 is an UNICODE encoding which uses a 16 bit word as the smallest unit. It is optimized to the usage of the most often used characters.

6.3 UTF-32

This is the most simple UNICODE encoding, since it uses 32 bits for each character, and so does not need a variable number of bytes for each character. Each UNICODE character can be directly encoded in one 32 bit word.

7 The locale

Especially on UNIX, the proper setting of the correct locale is very important to achieve a correct behaviour of the application. Basically, the locale is defined by a set of environment variables which are shown in Table 2.

Table 2: LC_ environment variables

Environment variable	Affected operations
LANG	
LC_CTYPE	
LC_NUMERIC	
LC_TIME	
LC_COLLATE	
LC_MONETARY	
LC_MESSAGES	
LC_PAPER	
LC_NAME	
LC_ADDRESS	
LC_TELEPHONE	
LC_MEASUREMENT	
LC_IDENTIFICATION	
LC_ALL	

The important thing here is to note that the locale settings themselves are *not* inherited from the parent process when a new process is created. Only the environment variables are inherited, but they *do not* have a direct impact on the character functions within the C library. Instead, the application needs to set the locale to use through a specific function call:

```
char *setlocale(int category, const char *locale);
```

The category is a constant which describes the LC_ category to set, such as LC_CTYPE. The locale is a string which contains the locale to set, such as "de_DE.UTF-8". As a special case, an empty string can be passed as locale, which has the effect that the category is set from the corresponding environment variable.

If the application does not issue this call, the "C" locale is set by default, which is a portable locale using the ASCII character set. To make the application use the locale from the environment variables (and therefore portable to all locales), `setlocale` can be called like this:

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

8 Character encoding in C

8.1 The char data type

In ISO C and ANSI C, there is the usual char data type which can be used to store a single character. A pointer to char points to a sequence of characters. Functions like **printf()** can be used to print zero terminated character sequences (aka strings):

```
#include <stdio.h>

main() {
    const char *str = "Hallo char";
    printf(str);
}
```

According to the ISO standard, the **char** datatype has a size of 1, i.e. **sizeof(char)** always returns 1.

8.2 The wchar_t data type

ISO C (99?) and ANSI C also define the wide character data type wchar_t to store multibyte characters. As such, it is usually larger than a char, but the standard does not require it. For example, the GNU C Compiler uses 32 bits:

```
$ cat charsets.c
#include <stdio.h>

main() {
  printf("wchar: %d\n", sizeof(wchar_t));
}

$ ./charsets
wchar: 4
```

To create wide character literals and wide character string literals, the modified **L** is used:

```
const wchar_t wide = L'X';
const wchar_t* wideStr = L"Hello World";
```

8.2.1 Wide character string handling

Since the usual string functions like `strlen` or `strcat` work on char types, they can not be used to deal with wide characters. Special functions exist to deal with wide characters. The functions are defined in `wchar.h`. The most common string functions are shown in Table 3.

8.2.2 Wide character I/O

The same applies for the output functions like **printf()**. Similar to the string functions, the usual stdio functions can not be used with wide character strings, and therefore special functions exist which are defined in **stdio.h**.

8.2.3 Character conversion functions

In addition to the wide equivalents to already existing C functions, a number of new functions exist which primarily deal with character conversion:
Note that the behaviour of the character conversion functions depend on the setting of LC_CTYPE.

8.3 Example code

Table 3: C string functions and their wide string counterpart

C string function	Wide string function
`strlen()`	`wcslen()`
`strcpy()`	`wcscpy()`
`strncpy()`	`wcsncpy()`
`strcat()`	`wcscat()`
`strncat()`	`wcsncat()`
`strcmp()`	`wcscmp()`
`strncmp()`	`wcsncmp()`
`strchr()`	`wcschr()`
`strstr()`	`wcsstr()`
`strtok()`	`wcstok()`
`strtoll()`	`wcstoll()`
`strtol()`	`wcstol()`
`atol()`	`?????`
`atoi()`	`?????`

Table 4: C I/O functions and their wide string counterpart

C I/O function	Wide string I/O function
`printf()`	`wprintf()`
`sprintf()`	`swprintf()`
`fprintf()`	`fwprintf()`
`fputc()`	`fputwc()`
`putchar()`	`putwchar()`
`---`	`fwide()`

8.3.1 String handling

The following is a code example which uses some wide character functions:

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

int main() {
    const wchar_t *str1 = L"Hello";
    const wchar_t *str2 = L"World";
    wchar_t str3[100];

    wcscpy(str3, str1);
    wcscat(str3, str2);
    wprintf(L"%ls + %ls = %ls\n", str1, str2, str3);

    return 0;
}
```

```
$ gcc -o widestr widestr.c

$ ./widestr
Hello + World = HelloWorld
```

Table 5: C character conversion functions

C function	What it does
`size_t mbrlen(const char *s, size_t n, mbstate_t *ps);`	determine number of bytes in next multibyte character
`wint_t btowc(int c)`	Converts a single byte character to a wide character. NOTE: Use mbtowc instead.
`int mbtowc(wchar_t *pwc, const char *s, size_t n);`	Converts a multibyte sequence to a wide character
`size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);`	Multithread safe variant of `mbtowc()`
`int wctomb(char *s, wchar_t wc);`	Converts wide character wc to multibyte sequence s.
`size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);`	Multithread safe variant of wctomb.
`size_t mbsrtowcs(wchar_t *dest, const char **src, size_t len, mbstate_t *ps);`	Converts a multibyte string to a wide character string
`size_t wcsrtombs(char *dest, const wchar_t **src, size_t len, mbstate_t *ps);`	Converts a wide character string to a multibyte string
`size_t mbsnrtowcs(wchar_t *dest, const char **src, size_t nms, size_t len, mbstate_t *ps);`	Converts a multibyte string to a wide character string
`size_t wcsnrtombs(char *dest, const wchar_t **src, size_t nwc, size_t len, mbstate_t *ps);`	Convert a wide character string to a multibyte string

8.3.2 Character conversion

The following is a code example which uses some character conversion functions:

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

int main() {
  const wchar_t *str = L"Hello";
  char mbs[100] = {0};
  mbstate_t shiftState = {0};
  size_t result = wcsrtombs(mbs, &str, sizeof(mbs), &shiftState);

  printf("%d bytes written.", result);

  return 0;
}
```

```
$ gcc -o conv1 conv1.c
```

```
$ ./conv1
5 bytes written.
```

Now, this is not surprising, since all characters of the string are part of the old seven bit ASCII character set and can therefore be encoded into one byte each. Lets try something else; we change one of the letters of the string to a german umlaut and add some error handling:

```
/**
 * conv2.c - Another multi byte conversion test
 */
#include <errno.h>
#include <wchar.h>
#include <stdio.h>

int main() {
  const wchar_t *str = L"Hellö";
  char mbs[100] = {0};
  mbstate_t shiftState = {0};
  size_t result = wcsrtombs(mbs, &str, sizeof(mbs), &shiftState);

  if (result == (size_t)-1) {
    perror("wcsrtombs");
  } else {
    printf("%d bytes written.", result);
  }

  return 0;
}
```

```
$ gcc -o conv2 conv2.c

$ ./conv2
wcsrtombs: Invalid or incomplete multibyte or wide character
```

Remembering the sentence about locales not being inherited, it becomes clear why this happens. Even if we properly set the LC_ environment variables to an UTF-8 encoding, the C library function fails because it uses the "C" locale by default with a 7 bit ASCII encoding, where the german umlaut does not have a representation. We change the program to make it portable across locales:

```
/**
 * conv2b.c - Conversion program portable across locales.
 */
#include <locale.h>
#include <errno.h>
#include <wchar.h>
#include <stdio.h>

int main() {
  const wchar_t *str = L"Hellö";
  char mbs[100] = {0};
  mbstate_t shiftState = {0};
  size_t result = 0;

  setlocale(LC_ALL, "");

  result = wcsrtombs(mbs, &str, sizeof(mbs), &shiftState);

  if (result == (size_t)-1) {
    perror("wcsrtombs");
  } else {
    printf("%d bytes written.", result);
  }

  return 0;
}
```

```
$ gcc -o conv2b conv2b.c

$ export LC_CTYPE="de_DE.iso885915@euro"

$ ./conv2b
5 bytes written.
```

The five characters from the string have been converted to 5 bytes, which is natural since we configured an ISO 8859-15 character encoding. Lets change the locale to an UTF-8 encoding:

```
$ export LC_CTYPE="de_DE.UTF-8"

$ ./conv2b
6 bytes written.
```

As we can see, the system now converts the string with 5 characters to 6 bytes! This is because in UNICODE, the german umlaut is encoded to a number which needs 2 bytes in UTF-8 representation.

8.3.3 "Hello World" in japanese

Using some web translator, we find out that the well known english words "Hello" and "World" translate to "こんにちは" and "ワールド" in japanese. These strings are encoded in UTF-8 as follows:
Hello = {0xe3, 0x81, 0x93, 0xe3, 0x82, 0x93, 0xe3, 0x81, 0xab, 0xe3, 0x81, 0xa1, 0xe3, 0x81, 0xaf}
World = {0xe3, 0x83, 0xaf, 0xe3, 0x83, 0xbc, 0xe3, 0x83, 0xab, 0xe3, 0x83, 0x89}
Lets assume that "Hello World" is also in japanese simply the concatenation of these words (and hope that it does not mean something completely different then ...). We can then implement a japanese hello world application as follows:

```
/**
 * HelloWorld.c - A "hello world" application in japanese.
 */

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

int main() {
  const char helloWorldUTF8[] =
        {0xe3, 0x81, 0x93, 0xe3, 0x82, 0x93, 0xe3, 0x81, 0xab, 0xe3, 0x81, 0xa1,
         0xe3, 0x81, 0xaf,
         0x20,
         0xe3, 0x83, 0xaf, 0xe3, 0x83, 0xbc, 0xe3, 0x83, 0xab, 0xe3, 0x83, 0x89,
         0x00};
  mbstate_t shiftState = {0};
  wchar_t helloWorldStr[100];

  setlocale(LC_ALL, "");

  /* convert UTF8 to wstring */
  const char *str = helloWorldUTF8;
  mbsrtowcs(helloWorldStr, &str, sizeof(helloWorldStr), &shiftState);

  /* print the wide string */
  wprintf(helloWorldStr);

  return 0;
}
```

```
$ gcc -o HelloWorld HelloWorld.c

$ ./HelloWorld
こんにちは ワールド
```

Of course it would be much better if we do not need to store the text string as character array, but could directly paste it into the source code. Recent C and C++ compilers allow source code to be in UTF-8 format, so we can simply try this:

```
/**
  * HelloWorld2.c - A better "hello world" application in japanese.
  */

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

int main() {
  wchar_t* helloWorldStr = L"こんにちは ワールド";

  setlocale(LC_ALL, "");

  /* print the string */
  wprintf(helloWorldStr);

  return 0;
}
```

```
$ gcc -o HelloWorld2 HelloWorld2.c

$ ./HelloWorld2
ãããã«ã¡ã¯ ã¯ã¼ã«ã
```

Well, this is not exactly what we expected! What happened? We have created a wchar_t* sequence by using the "L" modifier. The string itself is directly encoded in the source code as UTF-8 string. So, the compiler took the string and created a byte sequence from it - using each of the three characters which are necessary to encode one of the japanese characters as a wchar_t character. This is because the "L" modified does not perform any conversions on the string, i.e. it does *not* convert the characters of the UTF-8 string to wchar_t characters!

We can still implement a similar behaviour if we perform the conversion ourselves. We have to interpret the UTF-8 string inside the source code as a multi byte sequence and convert it to a wchar_t* string manually:

```
/**
 * HelloWorld2b.c - A better "hello world" application in japanese.
 */

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

int main() {
  const char* helloWorldStrUTF8 = "こんにちは ワールド";

  setlocale(LC_ALL, "");

  /* convert UTF8 to wstring */
  mbstate_t shiftState = {0};
  wchar_t helloWorldStr[100];
  const char *str = helloWorldStrUTF8;
  mbsrtowcs(helloWorldStr, &str, sizeof(helloWorldStr), &shiftState);

  /* print the string */
  wprintf(helloWorldStr);

  return 0;
}
```

```
$ gcc -o HelloWorld2b HelloWorld2b.c

$ ./HelloWorld2b
こんにちは ワールド
```

Note that creating localized string literals should seldom be necessary in an application. Usually such strings should be read from a database or from a properties file instead of being hard coded in the application.

9 Character encoding in ORACLE

When storing multi byte texts in an ORACLE database, several prerequisites must be met: at least, the database must be created as multi byte database (e.g. UTF-8) and the client character set must be set correctly through the `NLS_LANG` environment variable.

9.1 Setting up a multibyte database

Language and character encoding specific database parameters can be queried through the *nls_database_parameters* view:

```
SQL> select * from nls_database_parameters;

PARAMETER                      VALUE
------------------------------ ------------------------------
NLS_LANGUAGE                   AMERICAN
NLS_TERRITORY                  AMERICA
NLS_CURRENCY                   $
NLS_ISO_CURRENCY               AMERICA
NLS_NUMERIC_CHARACTERS         .,
NLS_CHARACTERSET               US7ASCII
NLS_CALENDAR                   GREGORIAN
NLS_DATE_FORMAT                DD-MON-YY
NLS_DATE_LANGUAGE              AMERICAN
NLS_SORT                       BINARY
NLS_TIME_FORMAT                HH.MI.SSXFF AM
NLS_TIMESTAMP_FORMAT           DD-MON-YY HH.MI.SSXFF AM
NLS_TIME_TZ_FORMAT             HH.MI.SSXFF AM TZH:TZM
NLS_TIMESTAMP_TZ_FORMAT        DD-MON-YY HH.MI.SSXFF AM TZH:T
NLS_DUAL_CURRENCY              $
```

```
NLS_COMP
NLS_NCHAR_CHARACTERSET          US7ASCII
NLS_RDBMS_VERSION               8.1.5.0.0

18 rows selected.
```

Especially important are the *NLS_CHARACTERSET* and the *NLS_NCHAR_CHARACTERSET* values. In the example above, both are set to *US7ASCII*.
Once the database is running in UTF8 mode, the proper setting of the *NLS_CHARACTERSET* parameters can be verified:

```
SQL> select * from nls_database_parameters where parameter like 'NLS_%CHARACTERSET';

    PARAMETER                      VALUE
    ------------------------------ ------------------------------
    NLS_CHARACTERSET               UTF8
    NLS_NCHAR_CHARACTERSET         UTF8

    SQL>
```

Lets create a table and insert a row into it:

```
SQL> create table t_i18ntext (c_oid integer, c_text varchar2(10));

Table created.

SQL> insert into t_i18ntext values(1, 'HelloWorld');

1 row created.

SQL> select * from t_i18ntext;

C_OID       C_TEXT
---------- ----------
         1 HelloWorld

SQL>
```

There is nothing unusal with this, as the text only contains seven bit ASCII characters. This example would also succeed with an US7ASCII database. Now, lets insert some japanese characters. We use an UNICODE xterm and we can use one of the sample programs from Section 8.3.3 to output an unicode text on the console, which can then be pasted into SQL*PLUS with copy&paste.

```
SQL> insert into t_i18ntext values(2, 'こんにちは ワールド');
insert into t_i18ntext values(2, 'こんにちは ワールド')

*
ERROR at line 1:
ORA-12899: value too large for column "ORCL"."T_I18NTEXT"."C_TEXT" (actual: 28, maximum: 1
SQL>
```

Even though the japanese text only contains 10 characters, it does not fit into the VARCHAR2(10) column! To be more precise, it would need 28 bytes (which are 9 * 3 = 27 characters for the encoding of the japanese characters plus one byte for the space between the two words).
This makes some things very complicated when migrating singlebyte applications to multibyte. To be sure that the number of characters which fitted into the column earlier still fit after the conversion to multibyte, each column needs to be made four times as large. For a VARCHAR2(2000), this would mean to convert it into

a VARCHAR2(8000) (which is in practice not possible with ORACLE, since a column can contain at most 4000 characters). Also, this issue does not only occur when talking about chinese and japanese characters (the issue could be less severe there, since in some of these languages a single character describes a whole word or at least a syllable). Even when german text with umlauts in UTF-8 encoding, each umlaut is encoded with two bytes, and the EURO sign needs three bytes.

To be more explicit: when migrating a singlebyte database to UTF8, the database schema must be adjusted. The length parameter used with VARCHAR2 defines bytes, not characters. The following example shows how we finally manage to store the japanese text in the database:

```
SQL> alter table t_i18ntext modify c_text varchar2(40);

Table altered.

SQL>  insert into t_i18ntext values(2, 'こんにちは ワールド');

1 row created.

SQL> select * from t_i18ntext;

C_OID      C_TEXT
---------- ----------------------------------------
         2 ccc+c!c/ c/c<c+c
```

Once again, this is not really what we expected. The issue here is that the client uses a different encoding than the server. The encoding used by the client can be changed by setting the `NLS_LANG` environment variable. It must be set for both the insert and the select operation, otherwise the data is not correctly inserted or selected.

```
$ export NLS_LANG=AMERICAN_AMERICA.UTF8

$ sqlplus scott/tiger

SQL*Plus: Release 8.1.5.0.0 - Production on Sat Jul 2 18:39:55 2005

SQL> insert into t_i18ntext values(3, 'こんにちは ワールド');

1 row created.

SQL> commit;

Commit complete.

SQL> select * from t_i18ntext;

C_OID      C_TEXT
---------- ----------------------------------------
         2 ccc+c!c/ c/c<c+c
         3 こんにちは ワールド

SQL>
```

This example shows that the data in the first row was already inserted wrong. Even though we set the `NLS_LANG` variable now, only the data from the second row has been correctly inserted and can also be retrieved again.

9.2 Accessing a multibyte database through OCI

This chapter describes how to access data in multibyte table columns from C through the Oracle Call Interface (OCI).

10 Character encoding in JAVA

Java uses UCS2 (16 bit UNICODE encoding) as internal strings representation, to handle all characters from UNICODE 2.1. Starting with Java 5, UNICODE 3.1 will be supported so that the complete UNICODE encoding space can be used.

11 Character encoding in C++

11.1 The wide string

Similar to the extension of the char datatype to wchar_t in C, C++ extends the STL string type to wstring.

```
typedef basic_string<wchar_t> wstring;
```

12 Migrating from singlebyte to UTF-8

Some things to think about when migrating old singlebyte database applications to multibyte applications:

```
- Network bandwith?
- Database size? double, triple, quadruple?
- memory footprint of the application? strings?
- database schema? varchar, nvarchar?
- Add NVARCHAR columns vs. converting all VARCHAR to multi byte (what abouth VARCHAR2(2000
```

Links to useful resources

[1] *The Unicode Home Page*, , <http://www.unicode.org> .

[2] *The UNICODE Howto*, , <http://www.faqs.org/docs/Linux-HOWTO/Unicode-HOWTO.html> .

[3] *Unicode and Multilingual Support in HTML, Fonts, Web Browsers and Other Applications*, , <http://www.alanwood.net/unicode/index.html> .

[4] *UTF-8 and Unicode FAQ for Unix/Linux*, , <http://www.cl.cam.ac.uk/~mgk25/unicode.html> .

[5] *yudit - An unicode text editor*, , <http://> .

[6] *Design a Single Unicode App that Runs on Both Windows 98 and Windows 2000*, , <http://www.microsoft.com/msj/defaultframe.asp?page=/msj/0499/multilangunicode .

[7] *glibc*, , <http://gnu.archive.hk/software/libc/manual/html_node/Extended-Char-Intro.
.

[8] *English to japanese translator*, , <http://www.suteki.nu/translator/> .