Encodings - Unicode - I18N

Foreword

[JLF 01/01/2017]
This document was written in 2003 during a past project to migrate a graph editor to Unicode.
The goal was to use UTF-16 strings internally (wchar*), instead of ANSI strings (char*), to use the TCHAR abstraction and to create windows that are registered as Unicode windows. It was then possible to display any Unicode string in the menus and in the graphs.

This document has not been updated since 2003 and may contain some obsolete informations.

Definitions

ACS : Abstract Character Set : Before assigning each character a number

CCS : Coded Character Set : After assigning each character a number. Mapping from integer numbers to character representations.

SBCS : The most common encodings (character encoding schemes) use a single byte per character, and they are often called single-byte character sets. They are all limited to 256 characters.

ASCII is a character set using 7-bit units, with a trivial encoding designed for 7-bit bytes

ISO-8859-1 : It is an 8-bit superset of ASCII and provides most of the characters necessary for Western Europe.

ISO-8859-15 : Modernized version of ISO-8859-1, with the euro symbol and some more French and Finnish letters.

DBCS : The Double-byte character sets were developed to provide enough space for the thousands of ideographic characters in East Asian writing systems. Here, the encoding is still byte-based, but each two bytes together represent a single character.
It's not possible to combine two languages, for example Japanese and Chinese, in the same data stream because the same double-byte code points represent different characters depending on the code page.

MBCS : Multi-byte character sets use a variable number of bytes per character, which distinguishes them from the DBCS encodings. MBCSs are often compatible with ASCII; that is, the Latin letters are represented in such encodings with the same bytes that ASCII uses. Some less often used characters may be encoded using three or even four bytes.

Unicode : The Unicode standard specifies a character set and several encodings.
The standard assigns numbers from 0 to 0x10FFFF, which is more than a million possible numbers for characters. The highest value, 0x10FFFF, takes up only 21 bits. 11 bits are always unused in a 32-bit word storing a Unicode code point.

Unicode provides a single character set that covers the languages of the world, and a small number of machine-friendly encoding forms and schemes to fit the needs of existing applications and protocols. It is designed for best interoperability with both ASCII and ISO-8859-1, the most widely used character sets, to make it easier for Unicode to be used in applications and protocols.

UCD : Unicode Character Database

Surrogate : Pair of Unicode value.

Glyph : A glyph is a particular image that represents a character or part of a character

Grapheme : What an end-user thinks of as a character.

Code point : What a character encoding standard encodes.

Code unit : A memory storage unit in a character encoding : 8, 16 or 32 bits.

Each UTF-n represents a code point as a sequence of one or more code units, where each code unit occupies n bits. There are three common ways to store Unicode strings:
- UTF-32, with 32-bit code units, each storing a single code point. It is very similar to the ISO 10646 format UCS-4, except that it is constrained to valid Unicode values for interoperability.
- UTF-16, with one or two 16-bit code units for each code point (this is the default encoding for Unicode)
- UTF-8, with one to four 8-bit code units (bytes) for each code point

For 16- and 32-bit code units it is important to know whether the most or least significant byte is written first. Thus, for byte streams, both UTF-16 and UTF-32 need to be specified as big-endian (most significant byte first) or little-endian (least significant byte first). Big-endian is the preferred network byte order as defined in Internet protocols. Microsoft uses little-endian...

SCSU : Standard Compression Scheme for Unicode

Character Encoding Forms : Internal encodings, with the byte ordering determined by the machine architecture.

CES : Character Encoding Schemes : External encodings for byte streams.
In UNIX the most-used CES is UTF-8. It allows for full support of the entire Unicode, all pages and planes, and will still read standard ASCII correctly. The alternatives to UTF-8 are: UCS-4, UTF-16, UTF-7,5, UTF-7, SCSU, HTML, and JAVA.

The Unicode CCS 3.1 is officially known as the ISO 10646-1 Universal Multiple Octet Coded Character Set (UCS).

The Unicode CCS utilizes a four-dimensional coding space of 128 three-dimensional groups.
Each group has 256 two-dimensional planes.
Each plane consists of 256 one-dimensional rows and each row has 256 cells.
A cell codes a character at this coding space or the cell is declared unused.
This coding concept is called UCS-4; four octets of bits are used to represent each character specifying the group, plane, row and cell.
128 groups * 256 planes * 256 rows * 256 cells

BMP : The first plane (plane 00 of the group 00) is the Basic Multilingual Plane (BMP).
The BMP defines characters in general use in alphabetic, syllabic and ideographic scripts as well as various symbols and digits. Subsequent planes are used for additional characters or other coded entities not yet invented. This full range is needed to cope with all of the world's languages; specifically, some East Asian languages that have almost 64,000 characters.

The BMP is used as a two-octet coded character set identified as the UCS-2 form of ISO 10646. ISO 10646 USC-2 is commonly referred to as (and is identical to) Unicode. This BMP, like all UCS planes, contains 256 rows each of 256 cells, and a character is coded at a cell by just the row and cell octets in the BMP. This allows 16-bit coding characters to be used for writing most commercially important languages. USC-2 requires no code page switching, code extensions or code states. USC-2 is a simple method to incorporate Unicode into software, but it is limited in only supporting the Unicode BMP.

[Microsoft] UCS-2
UCS-2 is the main Unicode encoding used by Microsoft Windows NT® 4.0, Microsoft® SQL Server™ version 7.0, and Microsoft SQL Server 2000. UCS-2 allows for encoding of 65,536 different code points. All information that is stored in Unicode in SQL Server 2000 is stored in this encoding, which uses two bytes for every character, regardless of the character being used.

[Microsoft] UTF-16
UTF-16 is the primary Unicode encoding used by Microsoft Windows 2000. Even before Unicode 2.0 was released, it became clear that the goal of Unicode (to support a single code point for every character in every language) could not be achieved using only 65,536 characters. Some languages, such as Chinese, require that many characters to encode just the rarely used characters. Thus, support was added for a surrogate range to handle an additional 1,048,576 characters. UTF-16 is the encoding that fully supports this extension to the original standard.
In UTF-16, the same standard of 2 bytes per code point is followed; however, with UTF-16 certain code points use another code point right after them to define the character.
Like UCS-2, UTF-16 is stored in a Little Endian manner, as is everything on Windows, by default.

[Microsoft] UTF-8
Many ASCII and other byte-oriented systems that require 8-bit encodings (such as mail servers) must span a vast array of computers that use different encodings, different byte orders, and different languages. UTF-8 is an encoding scheme that is designed to treat Unicode data in a way that is independent of the byte ordering on the computer. Although SQL Server 2000 does not store data in UTF-8 format, it supports UTF-8 in at least one crucial scenario: its support of the Extensible Markup Language (XML).
In the Windows environment, UTF-8 storage has these disadvantages:
- The Component Object Model (COM) supports only UTF-16/UCS-2 in its APIs and interfaces, which would require constant conversion if data were stored in UTF-8 format (this issue only applies when COM is used; SQL Server database engine does not typically call COM interfaces).
- The Windows NT and Windows 2000 kernels are both Unicode and use UCS-2 and UTF-16, respectively. Once again, a UTF-8 storage format would require many extra conversions (as with the previous note on COM, this would not result in a conversion hit in the SQL Server database engine, but would potentially affect many client-side operations).

- UTF-8 can be slower for many string operations. Sorting, comparing, and virtually any string operation can be slowed because characters do not have a fixed width.
- UTF-8 will often need more than 2 bytes, and the increased size can make for a larger footprint on disk and in memory.

Because XML is, among other things, a very important standard for communication over the Internet (which has a strong byte-oriented bias), its default to UTF-8 is something that can make a lot of sense.

Locales

The language determines the text and data formatting conventions, while the Country/Region determines the national conventions. Every language has a unique mapping, represented by "code pages," which includes characters other than those in the alphabet (such as punctuation marks and numbers). A code page is a character set and is related to the language. As such, a locale is a unique combination of language, Country/Region, and code page.

The locale and code page setting can be changed at run time by calling the setlocale function.

Locale

A locale is a set of user preference information related to the user's language and sublanguage. An example of a language is "French," where the sublanguage could be French as spoken in Canada, France, or Switzerland. Locale information includes currency symbol; date, time, and number formatting information; localized days of the week and months of the year; the standard abbreviation for the name of the country/region; and character encoding information. (For a more complete list see the NLSAPI specification.) Each Windows NT system has a default system locale and one user locale per user, which may be different from the default system locale. Both can be changed through the control panel.
Applications can specify a locale on a per-thread basis when calling APIs.

Character encoding

A character encoding (also called a code page) is a set of numeric values, or code points, that represents a group of alphanumeric characters, punctuation, and symbols. Single-byte character encodings use 8 bits to encode 256 different characters. On Windows, the first 128 characters of all code pages consist of the standard ASCII set of characters. The characters from code point 128 to 255 represent additional characters and vary depending on the set of scripts represented by the character encoding (for a complete listing of character sets tables see Developing International Software for Windows 95 and Windows NT, published by Microsoft Press). Double-byte character encodings on Windows, used for Asian languages, use 8 to 16 bits to encode each character. Computers exchange information encoded in character encodings and render it on screens using fonts.

Windows NT supports OEM character encodings (those originally designed for MS-DOS®), ANSI character encodings (those introduced with Windows® 3.1) and Unicode. Unicode is a 16-bit character encoding that encompasses most of the scripts in wide computer use today (for more information on Unicode see The Unicode Standard published by the Unicode Consortium or visit http://www.unicode.org/). Windows 2000 uses Unicode as its base character encoding, meaning that all strings passed around internally in the system, including strings in Windows resource (.res) files, are encoded in Unicode. Windows NT also supports ANSI character encodings. Each API that takes a string as a parameter has two entry points—an "A" or ANSI entry point and a "W" or wide-character (Unicode) entry point.
Windows NT supports additional code pages for translating data to and from Unicode, including Macintosh, EBCDIC, and ISO encodings. It also contains translation tables for the UTF-7 and UTF-8 standards, which are commonly used to send Unicode-based data across networks, in particular across the Internet.

Code pages

A *code page* is a character set, which can include numbers, punctuation marks, and other glyphs. Different languages and locales may use different code pages. For example, ANSI code page 1252 is used for English and most European languages; OEM code page 932 is used for Japanese Kanji.
A code page can be represented in a table as a mapping of characters to single-byte values or multibyte values. Many code pages share the ASCII character set for characters in the range 0x00 – 0x7F.
The Microsoft run-time library uses the following types of code pages:

- System-default ANSI code page. By default, at startup the run-time system automatically sets the multibyte code page to the system-default ANSI code page, which is obtained from the operating system. The call:
  ```
  setlocale ( LC_ALL, "" );
  ```
  also sets the locale to the system-default ANSI code page.
- Locale code page. The behavior of a number of run-time routines is dependent on the current locale setting, which includes the locale code page. (For more information, see Locale-Dependent Routines.) By default, all locale-dependent routines in the Microsoft run-time library use the code page that corresponds to the "C" locale. At run-time you can change or query the locale code page in use with a call to setlocale.
- Multibyte code page. The behavior of most of the multibyte-character routines in the run-time library depends on the current multibyte code page setting. By default, these routines use the system-default ANSI

code page. At run-time you can query and change the multibyte code page with _getmbcp and _setmbcp, respectively.

- The "C" locale is defined by ANSI to correspond to the locale in which C programs have traditionally executed. The code page for the "C" locale ("C" code page) corresponds to the ASCII character set. For example, in the "C" locale, **islower** returns true for the values 0x61 – 0x7A only. In another locale, **islower** may return true for these as well as other values, as defined by that locale.

Codepage Identifiers

Identifier	Name
037	IBM EBCDIC - U.S./Canada
437	OEM - United States
500	IBM EBCDIC - International
708	Arabic - ASMO 708
709	Arabic - ASMO 449+, BCON V4
710	Arabic - Transparent Arabic
720	Arabic - Transparent ASMO
737	OEM - Greek (formerly 437G)
775	OEM - Baltic
850	OEM - Multilingual Latin I
852	OEM - Latin II
855	OEM - Cyrillic (primarily Russian)
857	OEM - Turkish
858	OEM - Multlingual Latin I + Euro symbol
860	OEM - Portuguese
861	OEM - Icelandic
862	OEM - Hebrew
863	OEM - Canadian-French
864	OEM - Arabic
865	OEM - Nordic
866	OEM - Russian
869	OEM - Modern Greek
870	IBM EBCDIC - Multilingual/ROECE (Latin-2)
874	ANSI/OEM - Thai (same as 28605, ISO 8859-15)
875	IBM EBCDIC - Modern Greek
932	ANSI/OEM - Japanese, Shift-JIS
936	ANSI/OEM - Simplified Chinese (PRC, Singapore)
949	ANSI/OEM - Korean (Unified Hangeul Code)
950	ANSI/OEM - Traditional Chinese (Taiwan; Hong Kong SAR, PRC)
1026	IBM EBCDIC - Turkish (Latin-5)
1047	IBM EBCDIC - Latin 1/Open System
1140	IBM EBCDIC - U.S./Canada (037 + Euro symbol)
1141	IBM EBCDIC - Germany (20273 + Euro symbol)
1142	IBM EBCDIC - Denmark/Norway (20277 + Euro symbol)
1143	IBM EBCDIC - Finland/Sweden (20278 + Euro symbol)
1144	IBM EBCDIC - Italy (20280 + Euro symbol)
1145	IBM EBCDIC - Latin America/Spain (20284 + Euro symbol)
1146	IBM EBCDIC - United Kingdom (20285 + Euro symbol)
1147	IBM EBCDIC - France (20297 + Euro symbol)
1148	IBM EBCDIC - International (500 + Euro symbol)
1149	IBM EBCDIC - Icelandic (20871 + Euro symbol)
1200	Unicode UCS-2 Little-Endian (BMP of ISO 10646)
1201	Unicode UCS-2 Big-Endian
1250	ANSI - Central European
1251	ANSI - Cyrillic
1252	ANSI - Latin I
1253	ANSI - Greek
1254	ANSI - Turkish
1255	ANSI - Hebrew
1256	ANSI - Arabic
1257	ANSI - Baltic
1258	ANSI/OEM - Vietnamese

1361	Korean (Johab)
10000	MAC - Roman
10001	MAC - Japanese
10002	MAC - Traditional Chinese (Big5)
10003	MAC - Korean
10004	MAC - Arabic
10005	MAC - Hebrew
10006	MAC - Greek I
10007	MAC - Cyrillic
10008	MAC - Simplified Chinese (GB 2312)
10010	MAC - Romania
10017	MAC - Ukraine
10021	MAC - Thai
10029	MAC - Latin II
10079	MAC - Icelandic
10081	MAC - Turkish
10082	MAC - Croatia
12000	Unicode UCS-4 Little-Endian
12001	Unicode UCS-4 Big-Endian
20000	CNS - Taiwan
20001	TCA - Taiwan
20002	Eten - Taiwan
20003	IBM5550 - Taiwan
20004	TeleText - Taiwan
20005	Wang - Taiwan
20105	IA5 IRV International Alphabet No. 5 (7-bit)
20106	IA5 German (7-bit)
20107	IA5 Swedish (7-bit)
20108	IA5 Norwegian (7-bit)
20127	US-ASCII (7-bit)
20261	T.61
20269	ISO 6937 Non-Spacing Accent
20273	IBM EBCDIC - Germany
20277	IBM EBCDIC - Denmark/Norway
20278	IBM EBCDIC - Finland/Sweden
20280	IBM EBCDIC - Italy
20284	IBM EBCDIC - Latin America/Spain
20285	IBM EBCDIC - United Kingdom
20290	IBM EBCDIC - Japanese Katakana Extended
20297	IBM EBCDIC - France
20420	IBM EBCDIC - Arabic
20423	IBM EBCDIC - Greek
20424	IBM EBCDIC - Hebrew
20833	IBM EBCDIC - Korean Extended
20838	IBM EBCDIC - Thai
20866	Russian - KOI8-R
20871	IBM EBCDIC - Icelandic
20880	IBM EBCDIC - Cyrillic (Russian)
20905	IBM EBCDIC - Turkish
20924	IBM EBCDIC - Latin-1/Open System (1047 + Euro symbol)
20932	JIS X 0208-1990 & 0121-1990
20936	Simplified Chinese (GB2312)
21025	IBM EBCDIC - Cyrillic (Serbian, Bulgarian)
21027	Extended Alpha Lowercase
21866	Ukrainian (KOI8-U)
28591	ISO 8859-1 Latin I
28592	ISO 8859-2 Central Europe
28593	ISO 8859-3 Latin 3
28594	ISO 8859-4 Baltic
28595	ISO 8859-5 Cyrillic
28596	ISO 8859-6 Arabic
28597	ISO 8859-7 Greek

28598	ISO 8859- Hebrew
28599	ISO 8859-9 Latin 5
28605	ISO 8859-15 Latin 9
29001	Europa 3
38598	ISO 8859- Hebrew
50220	ISO 2022 Japanese with no halfwidth Katakana
50221	ISO 2022 Japanese with halfwidth Katakana
50222	ISO 2022 Japanese JIS X 0201-1989
50225	ISO 2022 Korean
50227	ISO 2022 Simplified Chinese
50229	ISO 2022 Traditional Chinese
50930	Japanese (Katakana) Extended
50931	US/Canada and Japanese
50933	Korean Extended and Korean
50935	Simplified Chinese Extended and Simplified Chinese
50936	Simplified Chinese
50937	US/Canada and Traditional Chinese
50939	Japanese (Latin) Extended and Japanese
51932	EUC - Japanese
51936	EUC - Simplified Chinese
51949	EUC - Korean
51950	EUC - Traditional Chinese
52936	HZ-GB2312 Simplified Chinese
54936	**Windows XP:** GB18030 Simplified Chinese (4 Byte)
57002	ISCII Devanagari
57003	ISCII Bengali
57004	ISCII Tamil
57005	ISCII Telugu
57006	ISCII Assamese
57007	ISCII Oriya
57008	ISCII Kannada
57009	ISCII Malayalam
57010	ISCII Gujarati
57011	ISCII Punjabi
65000	Unicode -7
65001	Unicode

GetCPInfo

The **GetCPInfo** function retrieves information about any valid installed or available code page.
To obtain additional information about valid installed or available code pages, use the **GetCPInfoEx** function.

```
BOOL GetCPInfo(
  UINT CodePage,      // code page identifier
  LPCPINFO lpCPInfo   // information buffer
);
```

Parameters

CodePage

[in] Specifies the code page about which information is to be retrieved. You can specify the code page identifier for any installed or available code page, or you can specify one of the following predefined values.

Value	Meaning
CP_ACP	Use the system default–ANSI code page.
CP_MACCP	**Windows NT/2000/XP:** Use the system default–Macintosh code page.
CP_OEMCP	Use the system default–OEM code page.
CP_THREAD_ACP	**Windows 2000/XP:** Use the current thread's ANSI code page.

See Code Page Identifiers for a list of ANSI and other code pages.

Windows 95/98/Me: The **GetCPInfo** version supported by the Microsoft Layer for Unicode also supports CP_UTF7 and CP_UTF8.

lpCPInfo

[out] Pointer to a **CPINFO** structure that receives information about the code page.

Return Values

If the function succeeds, the return value is 1.
If the function fails, the return value is zero. To get extended error information, call **GetLastError**.
If the specified code page is not installed or not available, **GetCPInfo** sets the last-error value to ERROR_INVALID_PARAMETER.

setlocale

Define the locale.

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

Return Value
If a valid locale and category are given, returns a pointer to the string associated with the specified locale and category. If the locale or category is invalid, returns a null pointer and the current locale settings of the program are not changed.
For example, the call

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

sets all categories, returning only the string English_USA.1252. If all categories are not explicitly set by a call to **setlocale**, the function returns a string indicating the current setting of each of the categories, separated by semicolons. If the *locale* argument is a null pointer, **setlocale** returns a pointer to the string associated with the *category* of the program's locale; the program's current locale setting is not changed.

The null pointer is a special directive that tells **setlocale** to query rather than set the international environment. For example, the sequence of calls

```
// Set all categories and return "English_USA.1252"
setlocale( LC_ALL, "English" );
// Set only the LC_MONETARY category and return "French_France.1252"
setlocale( LC_MONETARY, "French" );
setlocale( LC_ALL, NULL );
```

returns

```
LC_COLLATE=English_USA.1252;
LC_CTYPE=English_USA.1252;
LC_MONETARY=French_France.1252;
LC_NUMERIC=English_USA.1252;
LC_TIME=English_USA.1252
```

which is the string associated with the **LC_ALL** category.
You can use the string pointer returned by **setlocale** in subsequent calls to restore that part of the program's locale information, assuming that your program does not alter the pointer or the string. Later calls to **setlocale** overwrite the string; you can use _strdup to save a specific locale string.
Remarks
Use the **setlocale** function to set, change, or query some or all of the current program locale information specified by *locale* and *category*. *locale* refers to the locality (country/region and language) for which you can customize certain aspects of your program. Some locale-dependent categories include the formatting of dates and the display format for monetary values. If you set *locale* to the default string for a language with multiple forms supported on your computer, you should check the **setlocale** return code to see which language is in effect. For example, using "chinese" could result in a return value of **chinese-simplified** or **chinese-traditional**.

The *category* argument specifies the parts of a program's locale information that are affected. The macros used for *category* and the parts of the program they affect are as follows:

LC_ALL

All categories, as listed below.

LC_COLLATE

The **strcoll**, **_stricoll**, **wcscoll**, **_wcsicoll**, **strxfrm**, **_strncoll**, **_strnicoll**, **_wcsncoll**, **_wcsnicoll**, and **wcsxfrm** functions.

LC_CTYPE

The character-handling functions (except **isdigit**, **isxdigit**, **mbstowcs**, and **mbtowc**, which are unaffected).

LC_MONETARY

Monetary-formatting information returned by the **localeconv** function.

LC_NUMERIC

Decimal-point character for the formatted output routines (such as **printf**), for the data-conversion routines, and for the nonmonetary-formatting information returned by **localeconv**. In addition to the decimal-point character, **LC_NUMERIC** also sets the thousands separator and the grouping control string returned by localeconv.

LC_TIME

The **strftime** and **wcsftime** functions.

The *locale* argument is a pointer to a string that specifies the name of the locale. If *locale* points to an empty string, the locale is the implementation-defined native environment. A value of *C* specifies the minimal ANSI conforming environment for C translation. The *C* locale assumes that all **char** data types are 1 byte and that their value is always less than 256. The *C* locale is the only locale supported in Microsoft Visual C++ version 1.0 and earlier versions of Microsoft C/C++. At program startup, the equivalent of the following statement is executed:

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

The *locale* argument takes the following form:

```
locale :: "lang[_country_region[.code_page]]"
            | ".code_page"
            | ""
            | NULL
```

If *locale* is a null pointer, **setlocale** queries, rather than sets, the international environment, and returns a pointer to the string associated with the specified *category*. The program's current locale setting is not changed. For example,

```
setlocale( LC_ALL, NULL );
```

returns the string associated with *category*.
The following examples pertain to the **LC_ALL** category. Either of the strings ".OCP" and ".ACP" can be used in place of a code page number to specify use of the user-default OEM code page and user-default ANSI code page, respectively.

setlocale(LC_ALL, "");

Sets the locale to the default, which is the user-default ANSI code page obtained from the operating system.

setlocale(LC_ALL, ".OCP");

Explicitly sets the locale to the current OEM code page obtained from the operating system.

setlocale(LC_ALL, ".ACP");

Sets the locale to the ANSI code page obtained from the operating system.

setlocale(LC_ALL, "[lang_ctry]");

Sets the locale to the language and country/region indicated, using the default code page obtained from the host operating system.

setlocale(LC_ALL, "[lang_ctry.cp]");

Sets the locale to the language, country/region, and code page indicated in the [*lang_ctry.cp*] string. You can use various combinations of language, country/region, and code page. For example:

```
setlocale( LC_ALL, "French_Canada.1252" );
// Set code page to French Canada ANSI default
setlocale( LC_ALL, "French_Canada.ACP" );
```

```
// Set code page to French Canada OEM default
setlocale( LC_ALL, "French_Canada.OCP" );
```

setlocale(LC_ALL, "[lang]");

Sets the locale to the country/region indicated, using the default country/region for the language specified, and the user-default ANSI code page for that country/region as obtained from the host operating system. For example, the following two calls to **setlocale** are functionally equivalent:

```
setlocale( LC_ALL, "English" );
setlocale( LC_ALL, "English_United States.1252" );
```

setlocale(LC_ALL, "[.code_page]");

Sets the code page to the value indicated, using the default country/region and language (as defined by the host operating system) for the specified code page.

The category must be either **LC_ALL** or **LC_CTYPE** to effect a change of code page. For example, if the default country/region and language of the host operating system are "United States" and "English," the following two calls to **setlocale** are functionally equivalent:

```
setlocale( LC_ALL, ".1252" );
setlocale( LC_ALL, "English_United States.1252");
```

Preprocessor directive

#pragma setlocale("[*locale-string*]**")**

Defines the locale (Country/Region and language) to be used when translating wide-character constants and string literals. Since the algorithm for converting multibyte characters to wide characters may vary by locale or the compilation may take place in a different locale from where an executable file will be run, this pragma provides a way to specify the target locale at compile time. This guarantees that the wide-character strings will be stored in the correct format.

The default *locale-string* is "".

The "C" locale maps each character in the string to its value as a **wchar_t** (unsigned short). Other values that are valid for **setlocale** are those entries that are found in the Language Strings list. For example, you could issue:

```
#pragma setlocale("dutch")
```

The ability to issue a language string depends on the code page and language ID support on your computer.

Language strings recognized by setlocale

The following language strings are recognized by **setlocale**. Any language not supported by the operating system is not accepted by **setlocale**. The three-letter language-string codes are only valid in Windows 98, Windows Me, Windows NT, Windows 2000, and Windows XP.

Primary language	Sublanguage	Language string
Chinese	Chinese	"chinese"
Chinese	Chinese (simplified)	"chinese-simplified" or "chs"
Chinese	Chinese (traditional)	"chinese-traditional" or "cht"
Czech	Czech	"csy" or "czech"
Danish	Danish	"dan" or "danish"
Dutch	Dutch (default)	"dutch" or "nld"
Dutch	Dutch (Belgian)	"belgian", "dutch-belgian", or "nlb"
English	English (default)	"english"
English	English (Australian)	"australian", "ena", or "english-aus"
English	English (Canadian)	"canadian", "enc", or "english-can"
English	English (New Zealand)	"english-nz" or "enz"
English	English (United Kingdom)	"eng", "english-uk", or "uk"
English	English (United States)	"american", "american english", "american-english", "english-american", "english-us", "english-usa", "enu", "us", or "usa"
Finnish	Finnish	"fin" or "finnish"
French	French (default)	"fra" or "french"
French	French (Belgian)	"frb" or "french-belgian"
French	French (Canadian)	"frc" or "french-canadian"
French	French (Swiss)	"french-swiss" or "frs"
German	German (default)	"deu" or "german"

German	German (Austrian)	"dea" or "german-austrian"
German	German (Swiss)	"des", "german-swiss", or "swiss"
Greek	Greek	"ell" or "greek"
Hungarian	Hungarian	"hun" or "hungarian"
Icelandic	Icelandic	"icelandic" or "isl"
Italian	Italian (default)	"ita" or "italian"
Italian	Italian (Swiss)	"italian-swiss" or "its"
Japanese	Japanese	"japanese" or "jpn"
Korean	Korean	"kor" or "korean"
Norwegian	Norwegian (default)	"norwegian"
Norwegian	Norwegian (Bokmal)	"nor" or "norwegian-bokmal"
Norwegian	Norwegian (Nynorsk)	"non" or "norwegian-nynorsk"
Polish	Polish	"plk" or "polish"
Portuguese	Portuguese (default)	"portuguese" or "ptg"
Portuguese	Portuguese (Brazilian)	"portuguese-brazil" or "ptb"
Russian	Russian (default)	"rus" or "russian"
Slovak	Slovak	"sky" or "slovak"
Spanish	Spanish (default)	"esp" or "spanish"
Spanish	Spanish (Mexican)	"esm" or "spanish-mexican"
Spanish	Spanish (Modern)	"esn" or "spanish-modern"
Swedish	Swedish	"sve" or "swedish"
Turkish	Turkish	"trk" or "turkish"

Country/Region strings recognized by setlocale

The following is a list of Country/Region strings recognized by **setlocale**. Strings for countries/regions that are not supported by the operating system are not accepted by **setlocale**. Three-letter Country/Region-name codes are from the International Organization for Standardization and International Electrotechnical Commission (ISO/IEC) specification 3166.

Country/Region	Country/Region string
Australia	"aus" or "australia"
Austria	"aut" or "austria"
Belgium	"bel" or "belgium"
Brazil	"bra" or "brazil"
Canada	"can" or "canada"
China	"china", "chn", "pr china", or "pr-china"
Czech Republic	"cze" or "czech"
Denmark	"dnk" or "denmark"
Finland	"fin" or "finland"
France	"fra" or "france"
Germany	"deu" or "germany"
Greece	"grc" or "greece"
Hong Kong SAR	"hkg", "hong kong", or "hong-kong"
Hungary	"hun" or "hungary"
Iceland	"iceland" or "isl"
Ireland	"irl" or "ireland"
Italy	"ita" or "italy"
Japan	"jpn" or "japan"
Korea	"kor" or "korea"
Mexico	"mex" or "mexico"
The Netherlands	"nld", "holland", or "netherlands"
New Zealand	"nzl", "new zealand", "new-zealand", or "nz"
Norway	"nor" or "norway"
Poland	"pol" or "poland"
Portugal	"prt" or "portugal"
Russia	"rus" or "russia"
Singapore	"sgp" or "singapore"
Slovakia	"svk" or "slovak"
Spain	"esp" or "spain"
Sweden	"swe" or "sweden"
Switzerland	"che" or "switzerland"
Taiwan	"twn" or "taiwan"

Turkey	"tur" or "turkey"
United Kingdom	"gbr", "britain", "england", "great britain", "uk", "united kingdom", or "united-kingdom"
United States	"usa", "america", "united states", "united-states", or "us"

_setmbcp

Sets a new multibyte code page.

```
int _setmbcp(
   int codepage
);
```

Return Value

Returns 0 if the code page is set successfully. If an invalid code page value is supplied for *codepage*, returns –1 and the code page setting is unchanged.

Remarks

The **_setmbcp** function specifies a new multibyte code page. By default, the run-time system automatically sets the multibyte code page to the system-default ANSI code page. The multibyte code page setting affects all multibyte routines that are not locale dependent. However, it is possible to instruct **_setmbcp** to use the code page defined for the current locale (see the following list of manifest constants and associated behavior results). The multibyte code page also affects multibyte-character processing by the following run-time library routines:

_exec functions	_mktemp	_stat
_fullpath	_spawn functions	_tempnam
_makepath	_splitpath	tmpnam

In addition, all run-time library routines that receive multibyte-character *argv* or *envp* program arguments as parameters (such as the **_exec** and **_spawn** families) process these strings according to the multibyte code page. Hence, these routines are also affected by a call to **_setmbcp** that changes the multibyte code page.

The *codepage* argument can be set to any of the following values:

- **_MB_CP_ANSI** Use ANSI code page obtained from operating system at program startup.
- **_MB_CP_LOCALE** Use the current locale's code page obtained from a previous call to setlocale.
- **_MB_CP_OEM** Use OEM code page obtained from operating system at program startup. OEM code page is used by MS-DOS applications.
- **_MB_CP_SBCS** Use single-byte code page. When the code page is set to **_MB_CP_SBCS**, a routine such as _ismbblead always returns false. .
- Any other valid code page value, regardless of whether the value is an ANSI, OEM, or other operating-system-supported code page.

List of the multibyte routines that are dependent on the locale code page rather than the multibyte code page :

Routine	Use
mblen	Validate and return number of bytes in multibyte character
_mbstrlen	For multibyte-character strings: validate each character in string; return string length
mbstowcs	Convert sequence of multibyte characters to corresponding sequence of wide characters
mbtowc	Convert multibyte character to corresponding wide character
wcstombs	Convert sequence of wide characters to corresponding sequence of multibyte characters
wctomb	Convert wide character to corresponding multibyte character

Code Pages in Outlook 2002

The following table lists the values that are supported by the **InternetCodePage** property of Outlook 2002.

Name	Character Set	Code Page
Arabic (ISO)	iso-8859-6	28596
Arabic (Windows)	windows-1256	1256
Baltic (ISO)	iso-8859-4	28594
Baltic (Windows)	windows-1257	1257
Central European (ISO)	iso-8859-2	28592
Central European (Windows)	windows-1250	1250
Chinese Simplified (GB2312)	gb2312	936
Chinese Simplified (HZ)	hz-gb-2312	52936
Chinese Traditional (Big5)	big5	950
Cyrillic (ISO)	iso-8859-5	28595
Cyrillic (KOI8-R)	koi8-r	20866
Cyrillic (KOI8-U)	koi8-u	21866

```
Cyrillic (Windows)                windows-1251    1251
Greek (ISO)                       iso-8859-7      28597
Greek (Windows)                   windows-1253    1253
Hebrew (ISO-Logical)              iso-8859-8-i    38598
Hebrew (Windows)                  windows-1255    1255
Japanese (EUC)                    euc-jp          51932
Japanese (JIS)                    iso-2022-jp     50220
Japanese (JIS-Allow 1 byte Kana)  csISO2022JP     50221
Japanese (Shift-JIS)              iso-2022-jp     932
Korean                            ks_c_5601-1987  949
Korean (EUC)                      euc-kr          51949
Latin 3 (ISO)                     iso-8859-3      28593
Latin 9 (ISO)                     iso-8859-15     28605
Thai (Windows)                    windows-874     874
Turkish (ISO)                     iso-8859-9      28599
Turkish (Windows)                 windows-1254    1254
Unicode (UTF-7)                   utf-7           65000
Unicode (UTF-8)                   utf-8           65001
US-ASCII                          us-ascii        20127
Vietnamese (Windows)              windows-1258    1258
Western European (ISO)            iso-8859-1      28591
Western European (Windows)        Windows-1252    1252
```

The following table lists the code pages Microsoft recommends that you use for the best compatibility with older mail systems.

```
Name                          Character Set   Code Page
-----------------------------------------------------
US-ASCII                      us-ascii        20127
Western European (ISO)        iso-8859-1      28591
Central European (ISO)        iso-8859-2      28592
Cyrillic (KOI8-R)             koi8-r          20866
Cyrillic (Windows)            windows-1251    1251
Turkish (ISO)                 iso-8859-9      28599
Greek (ISO)                   iso-8859-7      28597
Baltic (ISO)                  iso-8859-4      28594
Hebrew (Windows)              windows-1255    1255
Arabic (Windows)              windows-1256    1256
Thai (Windows)                windows-874     874
Vietnamese (Windows)          windows-1258    1258
Japanese (JIS)                iso-2022-jp     50220
Chinese Simplified (GB2312)   gb2312          936
Korean                        ks_c_5601-1987  949
Chinese Traditional (Big5)    big5            950
Unicode (UTF-8)               utf-8           65001
```

Charsets in Microsoft Internet Explorer 5

The following table contains information about the character sets supported by Internet Explorer 5, and it includes the following information.

- **Charset Friendly Name**—Name used to refer to the character set.
- **Preferred Charset Label**— Most common identifier used to set character sets in Internet Explorer. For example, in the previous code sample the Charset Label is windows-1251. These identifiers are used for outbound data.
- **Aliases**— Other identifiers that can be used to set character sets. These identifiers are used for inbound data.
- **IE Ver**— Versions of Internet Explorer that support the listed character sets.
- **Min OS**— Minimum operating system that supports the listed character sets.
- **Code Page**— Code page that supports the listed character sets.
- **Family Code Page**— Indicates a Microsoft Windows® code page that is used to represent all or most of the characters in a charset.

Charset Friendly Name	Preferred Charset Label	Aliases	IE Ver	Min OS	Code Page	Family Code Page
Arabic (ASMO 708)	ASMO-708		IE5	Win95	708	1256
Arabic (DOS)	DOS-720		IE5	Win95	720	1256
Arabic (ISO)	iso-8859-6	arabic, csISOLatinArabic, ECMA-114, ISO_8859-6, ISO_8859-6:1987, iso-ir-127	IE5, IE4	Win95	28596	1256

Charset Friendly Name	Preferred Charset Label	Aliases	IE Ver	Min OS	Code Page	Family Code Page
Arabic (Mac)	x-mac-arabic		IE5	Win2000	10004	1256
Arabic (Windows)	windows-1256	cp1256	IE5	Win95	1256	1256
Baltic (DOS)	ibm775	CP500	IE5	Win2000	775	1257
Baltic (ISO)	iso-8859-4	csISOLatin4, ISO_8859-4, ISO_8859-4:1988, iso-ir-110, l4, latin4	IE5	Win95	28594	1257
Baltic (Windows)	windows-1257		IE5	Win95	1257	1257
Central European (DOS)	ibm852	cp852	IE5, IE4	Win95	852	1250
Central European (ISO)	iso-8859-2	csISOLatin2, iso_8859-2, iso_8859-2:1987, iso8859-2, iso-ir-101, l2, latin2	IE5, IE4	Win95	28592	1250
Central European (Mac)	x-mac-ce		IE5	Win2000	10029	1250
Central European (Windows)	windows-1250	x-cp1250	IE5	Win95	1250	1250
Chinese Simplified (EUC)	EUC-CN	x-euc-cn	IE5	Win2000	51936	936
Chinese Simplified (GB2312)	gb2312	chinese, CN-GB, csGB2312, csGB231280, csISO58GB231280, GB_2312-80, GB231280, GB2312-80, GBK, iso-ir-58	IE5, IE4	Win95	936	936
Chinese Simplified (HZ)	hz-gb-2312		IE5, IE4	Win95	52936	936
Chinese Simplified (Mac)	x-mac-chinesesimp		IE5	Win2000	10008	936
Chinese Traditional (Big5)	big5	cn-big5, csbig5, x-x-big5	IE5, IE4	Win95	950	950
Chinese Traditional (CNS)	x-Chinese-CNS		IE5	Win2000	20000	950
Chinese Traditional (Eten)	x-Chinese-Eten		IE5	Win2000	20002	950
Chinese Traditional (Mac)	x-mac-chinesetrad		IE5	Win2000	10002	950
Cyrillic (DOS)	cp866	ibm866	IE5, IE4	Win95	866	1251
Cyrillic (ISO)	iso-8859-5	csISOLatin5, csISOLatinCyrillic, cyrillic, ISO_8859-5, ISO_8859-5:1988, iso-ir-144, l5	IE5, IE4	Win95	28595	1251
Cyrillic (KOI8-R)	koi8-r	csKOI8R, koi, koi8, koi8r	IE5, IE4	Win95	20866	1251
Cyrillic (KOI8-U)	koi8-u	koi8-ru	IE5	Win95	21866	1251
Cyrillic (Mac)	x-mac-cyrillic		IE5	Win2000	10007	1251
Cyrillic (Windows)	windows-1251	x-cp1251	IE5	Win95	1251	1251
Europa	x-Europa		IE5	n.a.	29001	1252
German (IA5)	x-IA5-German		IE5	Win2000	20106	1252
Greek (DOS)	ibm737		IE5	Win2000	737	1253
Greek (ISO)	iso-8859-7	csISOLatinGreek, ECMA-118, ELOT_928, greek, greek8, ISO_8859-7, ISO_8859-7:1987, iso-ir-126	IE5, IE4	Win95	28597	1253
Greek (Mac)	x-mac-greek		IE5	Win2000	10006	1253
Greek (Windows)	windows-1253		IE5	Win95	1253	1253
Greek, Modern (DOS)	ibm869		IE5	Win2000	869	1253
Hebrew (DOS)	DOS-862		IE5	Win95	862	1255
Hebrew (ISO-Logical)	iso-8859-8-i	logical	IE5, IE4	Win95	38598	1255
Hebrew (ISO-Visual)	iso-8859-8	csISOLatinHebrew, hebrew, ISO_8859-8, ISO_8859-8:1988, ISO-8859-8, iso-ir-138, visual	IE5, IE4	Win95	28598	1255
Hebrew (Mac)	x-mac-hebrew		IE5	Win2000	10005	1255
Hebrew (Windows)	windows-1255	ISO_8859-8-I, ISO-8859-8, visual	IE5	Win95	1255	1255
IBM EBCDIC (Arabic)	x-EBCDIC-Arabic		IE5	Win2000	20420	1256
IBM EBCDIC (Cyrillic Russian)	x-EBCDIC-CyrillicRussian		IE5	Win2000	20880	1251
IBM EBCDIC (Cyrillic Serbian-Bulgarian)	x-EBCDIC-CyrillicSerbianBulgarian		IE5	Win2000	21025	1251
IBM EBCDIC (Denmark-Norway)	x-EBCDIC-DenmarkNorway		IE5	Win2000	20277	1252
IBM EBCDIC (Denmark-Norway-Euro)	x-ebcdic-denmarknorway-euro		IE5	Win2000	1142	1252
IBM EBCDIC (Finland-Sweden)	x-EBCDIC-FinlandSweden		IE5	Win2000	20278	1252
IBM EBCDIC (Finland-Sweden-Euro)	x-ebcdic-finlandsweden-euro		IE5	Win2000	1143	1252
IBM EBCDIC (Finland-Sweden-Euro)	x-ebcdic-finlandsweden-euro	X-EBCDIC-France	IE5	Win2000	1143	1252

Charset Friendly Name	Preferred Charset Label	Aliases	IE Ver	Min OS	Code Page	Family Code Page
IBM EBCDIC (France-Euro)	x-ebcdic-france-euro		IE5	Win2000	1147	1252
IBM EBCDIC (Germany)	x-EBCDIC-Germany		IE5	Win2000	20273	1252
IBM EBCDIC (Germany-Euro)	x-ebcdic-germany-euro		IE5	Win2000	1141	1252
IBM EBCDIC (Greek Modern)	x-EBCDIC-GreekModern		IE5	Win2000	875	1253
IBM EBCDIC (Greek)	x-EBCDIC-Greek		IE5	Win2000	20423	1253
IBM EBCDIC (Hebrew)	x-EBCDIC-Hebrew		IE5	Win2000	20424	1255
IBM EBCDIC (Icelandic)	x-EBCDIC-Icelandic		IE5	Win2000	20871	1252
IBM EBCDIC (Icelandic-Euro)	x-ebcdic-icelandic-euro		IE5	Win2000	1149	1252
IBM EBCDIC (International-Euro)	x-ebcdic-international-euro		IE5	Win2000	1148	1252
IBM EBCDIC (Italy)	x-EBCDIC-Italy		IE5	Win2000	20280	1252
IBM EBCDIC (Italy-Euro)	x-ebcdic-italy-euro		IE5	Win2000	1144	1252
IBM EBCDIC (Japanese and Japanese Katakana)	x-EBCDIC-JapaneseAndKana		IE5	Win2000	50930	932
IBM EBCDIC (Japanese and Japanese-Latin)	x-EBCDIC-JapaneseAndJapaneseLatin		IE5	Win2000	50939	932
IBM EBCDIC (Japanese and US-Canada)	x-EBCDIC-JapaneseAndUSCanada		IE5	Win2000	50931	932
IBM EBCDIC (Japanese katakana)	x-EBCDIC-JapaneseKatakana		IE5	Win2000	20290	932
IBM EBCDIC (Korean and Korean Extended)	x-EBCDIC-KoreanAndKoreanExtended		IE5	Win2000	50933	949
IBM EBCDIC (Korean Extended)	x-EBCDIC-KoreanExtended		IE5	Win2000	20833	949
IBM EBCDIC (Multilingual Latin-2)	CP870		IE5	Win2000	870	1250
IBM EBCDIC (Simplified Chinese)	x-EBCDIC-SimplifiedChinese		IE5	Win2000	50935	936
IBM EBCDIC (Spain)	X-EBCDIC-Spain		IE5	Win2000	20284	1252
IBM EBCDIC (Spain-Euro)	x-ebcdic-spain-euro		IE5	Win2000	1145	1252
IBM EBCDIC (Thai)	x-EBCDIC-Thai		IE5	Win2000	20838	874
IBM EBCDIC (Traditional Chinese)	x-EBCDIC-TraditionalChinese		IE5	Win2000	50937	950
IBM EBCDIC (Turkish Latin-5)	CP1026		IE5	Win2000	1026	1254
IBM EBCDIC (Turkish)	x-EBCDIC-Turkish		IE5	Win2000	20905	1254
IBM EBCDIC (UK)	x-EBCDIC-UK		IE5	Win2000	20285	1252
IBM EBCDIC (UK-Euro)	x-ebcdic-uk-euro		IE5	Win2000	1146	1252
IBM EBCDIC (US-Canada)	ebcdic-cp-us		IE5	Win2000	37	1252
IBM EBCDIC (US-Canada-Euro)	x-ebcdic-cp-us-euro		IE5	Win2000	1140	1252
Icelandic (DOS)	ibm861		IE5	Win2000	861	1252
Icelandic (Mac)	x-mac-icelandic		IE5	Win2000	10079	1252
ISCII Assamese	x-iscii-as		IE5	Win2000	57006	57006
ISCII Bengali	x-iscii-be		IE5	Win2000	57003	57003
ISCII Devanagari	x-iscii-de		IE5	Win2000	57002	57002
ISCII Gujarathi	x-iscii-gu		IE5	Win2000	57010	57010
ISCII Kannada	x-iscii-ka		IE5	Win2000	57008	57008
ISCII Malayalam	x-iscii-ma		IE5	Win2000	57009	57009
ISCII Oriya	x-iscii-or		IE5	Win2000	57007	57007
ISCII Panjabi	x-iscii-pa		IE5	Win2000	57011	57011
ISCII Tamil	x-iscii-ta		IE5	Win2000	57004	57004
ISCII Telugu	x-iscii-te		IE5	Win2000	57005	57005
Japanese (EUC)	euc-jp	csEUCPkdFmtJapanese, Extended_UNIX_Code_Packed_Format_for_Japane	IE5, IE4	Win95	51932	932

Charset Friendly Name	Preferred Charset Label	Aliases	IE Ver	Min OS	Code Page	Family Code Page
Japanese (JIS)	iso-2022-jp	se, x-euc, x-euc-jp	IE5, IE4	Win95	50220	932
Japanese (JIS-Allow 1 byte Kana - SO/SI)	iso-2022-jp	_iso-2022-jp$SIO	IE5	Win95	50222	932
Japanese (JIS-Allow 1 byte Kana)	csISO2022JP	_iso-2022-jp	IE5	Win95	50221	932
Japanese (Mac)	x-mac-japanese		IE5	Win2000	10001	932
Japanese (Shift-JIS)	shift_jis	csShiftJIS, csWindows31J, ms_Kanji, shift-jis, x-ms-cp932, x-sjis	IE5, IE4	Win95	932	932
Korean	ks_c_5601-1987	csKSC56011987, euc-kr, iso-ir-149, korean, ks_c_5601, ks_c_5601_1987, ks_c_5601-1989, KSC_5601, KSC5601	IE5	Win95	949	949
Korean (EUC)	euc-kr	csEUCKR	IE5	Win95	51949	949
Korean (ISO)	iso-2022-kr	csISO2022KR	IE5	Win95	50225	949
Korean (Johab)	Johab		IE5	Win2000	1361	1361
Korean (Mac)	x-mac-korean		IE5	Win2000	10003	949
Latin 3 (ISO)	iso-8859-3	csISO, Latin3, ISO_8859-3, ISO_8859-3:1988, iso-ir-109, l3, latin3	IE5, IE4	Win95	28593	1254
Latin 9 (ISO)	iso-8859-15	csISO, Latin9, ISO_8859-15, l9, latin9	IE5	Win95	28605	1252
Norwegian (IA5)	x-IA5-Norwegian		IE5	Win2000	20108	1252
OEM United States	IBM437	437, cp437, csPC8, CodePage437	IE5	Win2000	437	1252
Swedish (IA5)	x-IA5-Swedish		IE5	Win2000	20107	1252
Thai (Windows)	windows-874	DOS-874, iso-8859-11, TIS-620	IE5, IE4	Win95	874	874
Turkish (DOS)	ibm857		IE5	Win2000	857	1254
Turkish (ISO)	iso-8859-9	csISO, Latin5, ISO_8859-9, ISO_8859-9:1989, iso-ir-148, l5, latin5	IE5	Win95	28599	1254
Turkish (Mac)	x-mac-turkish		IE5	Win2000	10081	1254
Turkish (Windows)	windows-1254	ISO_8859-9, ISO_8859-9:1989, iso-8859-9, iso-ir-148, latin5	IE5	Win95	1254	1254
Unicode	unicode	utf-16	IE5, IE4	Win95	1200	1200
Unicode (Big-Endian)	unicodeFFFE		IE5, IE4	Win95	1201	1200
Unicode (UTF-7)	utf-7	csUnicode11UTF7, unicode-1-1-utf-7, x-unicode-2-0-utf-7	IE5, IE4	Win95	65000	1200
Unicode (UTF-8)	utf-8	unicode-1-1-utf-8, unicode-2-0-utf-8, x-unicode-2-0-utf-8	IE5, IE4	Win95	65001	1200
US-ASCII	us-ascii	ANSI_X3.4-1968, ANSI_X3.4-1986, ascii, cp367, csASCII, IBM367, ISO_646.irv:1991, ISO646-US, iso-ir-6us	IE5	Win95	20127	1252
Vietnamese (Windows)	windows-1258		IE5, IE4	Win95	1258	1258
Western European (DOS)	ibm850		IE5	Win2000	850	1252
Western European (IA5)	x-IA5		IE5	Win2000	20105	1252
Western European (ISO)	iso-8859-1	cp819, csISO, Latin1, ibm819, iso_8859-1, iso_8859-1:1987, iso8859-1, iso-ir-100, l1, latin1	IE5	Win95	28591	1252
Western European (Mac)	macintosh		IE5	Win2000	10000	1252
Western European (Windows)	Windows-1252	ANSI_X3.4-1968, ANSI_X3.4-1986, ascii, cp367, cp819, csASCII, IBM367, ibm819, ISO_646.irv:1991, iso_8859-1, iso_8859-1:1987, ISO646-US, iso8859-1, iso-8859-1, iso-ir-100, iso-ir-6, latin1, us, us-ascii, x-ansi	IE5	Win95	1252	1252

How to Encode XML Data

By Chris Lovett
Microsoft Corporation
March 2000
Summary: This article explains how character encoding works and specifically how it works in XML and the MSXML DOM.

A lot of people have been asking me questions lately about how to make their XML files transfer data properly between different platforms. They create an XML document, type in data, stick a few tags around it, make the tags well-formed, and even put the <?xml version="1.0"?> declaration in for good measure. Then they try and load it up but get an unexpected error message from the Microsoft® XML Parser (MSXML) saying that there's something wrong with their data. This can be frustrating to the new XML author. Shouldn't it just work?
Well, not quite. It's likely that when you receive the unexpected error message from MSXML, the platform that is receiving your data stores it differently than the platform from which you sent it, resulting in character encoding problems.

Cross-Platform Data Formats

Creating cross-platform technologies and allowing different platforms to share data is an area that computer software and hardware industries have struggled with ever since they managed to connect two computers together. Since the early days, things have only become more and more complex with the explosion in the number of different types of computers, different ways of connecting them, and different kinds of data that you might want to share between them.
After decades of research into cross-platform programming technologies, the only real cross-platform solution today (and probably for a long time to come) is that which is achieved by simple *standard data formats*. The success of the Web was built on exactly these formats. The main thing that passes between Web servers and Web browsers today is HTTP headers and HTML pages, both of which are standard text formats.
In the next few sections, I'll discuss character encoding and standard character sets, Unicode, the HTML Content-Type header, the HTML Content-Type metatags, and character entities. If you are familiar with these concepts, you can skip ahead to the tips and tricks of encoding XML data for the XML Document Object Model (DOM) programmer. For details, see XML and Character Encoding.

A Lesson in Character Encoding

Standard text formats are built on standard character sets. Remember that all computers store text as numbers. However, different systems can also store the same text using different numbers. The following table shows how a range of bytes is stored, first on a typical computer running Microsoft Windows® using the default code page 1252, and second on a typical Apple® Macintosh® computer using the Macintosh Roman code page.

Byte	Windows	Macintosh
140	Œ	å
229	å	Â
231	ç	Á
232	è	Ë
233	é	È

For example, when your grandma places an order for a new book from http://www.barnesandnoble.com/, she is unaware that her Macintosh computer stores its characters differently than the new Windows 2000 Web server running www.barnesandnoble.com. As she enters your Swedish home address in the ship-to field of the Internet order form, she believes the Internet will correctly deliver the character å (byte value 140 on her Macintosh), unaware that her message will be received and processed by computers that translate byte value 140 as the letter **Œ**.

Unicode

The Unicode Consortium decided it would be a good idea to define one universal code page (using 2 bytes instead of one per character) that covers all the languages of the world so that this mapping problem between different code pages would be gone forever.
So if Unicode solves cross-platform character encoding issues, why hasn't it become the only standard? The first problem is that switching to Unicode sometimes means doubling the size of all your files-which in a network-bound world is not ideal. Some people therefore still prefer to use the older, single-byte character sets such as ISO-8859-1 to ISO-8859-15, Shift-JIS, EUC-KR, and so forth.
The second problem is that there are still many systems out there that are not Unicode-based at all, which means that on a network, some of the byte values that make up the Unicode characters can cause major problems for those older

systems. So Unicode Transformation Formats (UTF) have been defined; they use bit-shifting techniques to encode Unicode characters as byte values that will be "transparent" (or flow through safely) on those older systems.
The most popular of these character encodings is UTF-8. UTF-8 takes the first 127 characters of the Unicode standard (which happen to be the basic Latin characters, A-Z, a-z, and 0-9, and a few punctuation characters) and maps those directly to single byte values. It then applies a bit-shifting technique using the high bit of the bytes to encode the rest of the Unicode characters. The result of all this is that the little Swedish character å (0xE5) becomes the following 2-byte gibberish Ã¥ (0xC3 0xA5). So unless you can do bit shifting in your head, data encoded in UTF-8 is not human readable.

Content-Type Header

Because the older single-byte character sets are still in use, the problem of transferring data is not solved until we also specify what actual character set the data is in. Recognizing this, the Internet e-mail and HTTP protocols groups defined a standard way to specify the character set in the message header **Content-Type** property. The property specifies a character set from the list of the registered character set names defined by the Internet Assigned Numbers Authority (IANA). A typical HTTP header might contain the following text:

```
HTTP/1.1 200 OK
Content-Length: 15327
Content-Type: text/html; charset:ISO-8859-1;
Server: Microsoft-IIS/5.0
Content-Location: http://www.microsoft.com/Default.htm
Date: Wed, 08 Dec 1999 00:55:26 GMT
Last-Modified: Mon, 06 Dec 1999 22:56:30 GMT
```

This header indicates to the application that what follows after the header is in the ISO-8859-1 character set.

Content-Type Metatags

The **Content-Type** property is optional, and in some applications, the HTTP header information is stripped off and just the HTML itself is passed along. To remedy this, the HTML standards group defined an optional metatag as a way to specify the character set in the HTML document itself, making the HTML document character set self-describing.

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-8859-1">
```

In this case, the character set ISO-8859-1 declares that in this particular HTML page, the byte value of 229 means å. This page is now completely unambiguous on any system, and data is not misinterpreted. Unfortunately, because this metatag is optional, it leaves room for error.

Character Entities

Not every system supports every registered character set. For example, I don't think many platforms actually support the IBM mainframe character set called EBCDIC. Windows NT does, but probably not many others-which is most likely why the http://www.ibm.com home page is generating ASCII.
As a backup plan, HTML allows the encoding of individual characters in the page by specifying their exact Unicode character value. These character entities are then parsed independently of the character set, and their Unicode values can be determined unambiguously. The syntax for this is "å" or "å".

XML and Character Encoding

XML borrowed these ideas from HTML and took them even further, defining a completely unambiguous algorithm to determine the character set encoding used. In XML, an optional encoding attribute on the XML declaration defines the character encoding. The following algorithm determines the default encodings:
If the file starts with a Unicode byte-order mark [0xFF 0xFE] or [0xFE 0xFF], the document is considered to be in UTF-16 encoding. Otherwise, it is in UTF-8.
The following are all correct and equivalent XML documents:

Character set or encoding	HTTP header	XML document
ISO-8859-1	Content-Type: text/xml; charset:ISO-8859-1;	<test>å</test>
UTF-8	Content-Type: text/xml;	<test>Ã¥</test>
ISO-8859-1	Content-Type: text/xml;	<?xml version="1.0" encoding="ISO-8859-1"?> <test>å</test>
UTF-8 (using character entities)	Content-Type: text/xml;	<test>å</test>
UTF-16 (Unicode with byte-order mark)	Content-Type: text/xml;	ff fe 3c 00 74 00 65 00 73 00 74 00 3e 00 e5 00 ..<.t.e.s.t.>... 3c 00 2f 00 74 00 65 00 73 00 74 00 3e 00 0d

00 <./.t.e.s.t.>...
0a 00

Character Sets and the MSXML DOM

Now that we've discussed various ways to encode characters, let's look at how to load XML documents in the MSXML DOM and the types of error messages you might get when encountering ambiguously-encoded characters. The two main methods of loading XML DOM documents are the **LoadXML** method and the **Load** method.
The **LoadXML** method always takes a Unicode BSTR that is encoded in UCS-2 or UTF-16 only. If you pass in anything other than a valid Unicode BSTR to **LoadXML,** it will fail to load.
The **Load** method can take the following as a VARIANT:

Value	Description
URL	If the VARIANT is a BSTR, it is interpreted as a URL.
VT_ARRAY | VT_UI1	The VARIANT can also be a SAFEARRAY containing the raw encoded bytes.
IUnknown	If the VARIANT is an **IUnknown** interface, the DOM document calls **QueryInterface** for **IStream, IPersistStream,** and **IPersistStreamInit.**

The **Load** method implements the following algorithm for determining the character encoding or character set of the XML document:

- If the Content-Type HTTP header defines a character set, this character set overrides anything in the XML document itself. This obviously doesn't apply to SAFEARRAY and **IStream** mechanisms because there is no HTTP header.
- If there is a 2-byte Unicode byte-order mark, it assumes the encoding is UTF-16. It can handle both big endian and little endian.
- If there is a 4-byte Unicode byte order mark (0xFF 0xFE 0xFF 0xFE), it assumes the encoding is UTF-32. It can handle both big endian and lttle endian.
- Otherwise, it assumes the encoding is UTF-8 unless it finds an XML declaration with an encoding attribute that specifies some other character set (such as ISO-8859-1, Windows-1252, Shift-JIS, and so on).

There are two errors you will see returned from the XML DOM that indicate encoding problems. The first usually indicates that a character in the document does not match the encoding of the XML document:

```
An invalid character was found in text content.
```

The **ParseError** object will tell you exactly where on a particular line this rogue character occurs so that you can fix the problem.
The second error indicates that you started off with a Unicode byte-order mark (or you called the **LoadXML** method), and then an encoding attribute specified something other than a 2-byte encoding (such as UTF-8 or Windows-1250):

```
Switch from current encoding to specified encoding not supported.
```

Alternatively, you could have called the **Load** method and started off with a single-byte encoding (no byte-order mark), but then it found an encoding attribute that specified a 2- or 4-byte encoding (such as UTF-16 or UCS-4).
The bottom line is that you cannot switch between a multibyte character set like UTF-8, Shift-JIS, or Windows-1250 and Unicode character encodings such as UTF-16, UCS-2, or UCS-4 using the encoding attribute on an XML declaration, because the declaration itself has to use the same number of bytes per character as the rest of the document.
Lastly, the **IXMLHttpRequest** interface provides the following ways of accessing downloaded data:

Methods	Description
ResponseXML	Represents the response entity body as parsed by the MSXML DOM parser, using the same rules as the **Load** method.
ResponseText	Represents the response entity body as a string. This method blindly decodes the received message body from UTF-8. This is a known problem that should be fixed in the upcoming MSXML Web Release.
ResponseBody	Represents the response entity body as an array of unsigned bytes.
ResponseStream	Represents the response entity body as an **IStream** interface.

Creating New XML Documents with MSXML

Once the XML document is loaded, you can manipulate that XML document using the DOM without concern for any encoding issues because the document is stored in memory as Unicode. All the XML DOM interfaces are based on COM BSTRs, which are 2-byte Unicode strings. This means you can build an MSXML DOM document from scratch in memory that contains all sorts of Unicode characters and all components will be able to share this DOM in memory without any confusion over the meaning of the Unicode character values. When you save this, however, MSXML will encode all data in UTF-8 by default. For example, suppose you do the following:

```
var xmldoc = new ActiveXObject("Microsoft.XMLDOM")
```

```
var e = xmldoc.createElement("test");
e.text = "å";
xmldoc.appendChild(e);
xmldoc.save("foo.xml");
```
The following UTF-8 encoded file will result:
<test>Ã¥</test>

> **Note** The preceding example will only work if you run the code outside the browser environment. Calling the **Save** method while inside the browser will not produce the same results because of security restrictions.

Even though this looks weird, it is correct. The following test loads up the UTF-8 encoded file and tests whether the UTF-8 is decoded back to the Unicode character value 229. It is:
```
var xmldoc = new ActiveXObject("Microsoft.XMLDOM")
xmldoc.load("foo.xml");
if (xmldoc.documentElement.text.charCodeAt(0) == 229)
{
    WScript.echo("Yippee - it worked !!");
}
```
To change the encoding that the XML DOM **Save** method uses, you need to create an XML declaration with an encoding attribute at the top of your document as follows:
```
var pi = xmldoc.createProcessingInstruction("xml",
                        " version='1.0' encoding='ISO-8859-1'");
xmldoc.appendChild(pi);
```
When you call the **save** method, you will then get an ISO-8859-1 encoded file as follows:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<test>å</test>
```
Now, be careful you don't let the **XML** property confuse you. The **XML** property returns a Unicode string. If you call the **XML** property on the **DOMDocument** object after creating the ISO-8859-1 encoding declaration, you will get the following Unicode string back:
```
<?xml version="1.0"?>
<test>å</test>
```
Notice that the ISO-8859-1 encoding declaration is gone. This is normal. The reason it did this is so that you can turn around and call **LoadXML** with this string and it will work. If it does not do this, **LoadXML** will fail with the error message: "Switch from current encoding to specified encoding not supported."

Conclusion

Hopefully this article has helped explain how character encoding works and specifically how it works in XML and the MSXML DOM. Character set encoding is pretty simple once you understand it, and XML is great because it leaves no room for ambiguity in this respect. The MSXML DOM has a few quirks to watch out for, but it remains a powerful tool that allows you to both read and write any XML encoding.

For More Information

- Microsoft MSDN Online Library: XML DOM Reference
- Character Encoding Model by Ken Whistler and Mark Davis
- IANA Character Sets
- Internet Engineering Task Force (IETF) at http://www.ietf.org for a list of RFCs
- Microsoft Global Software Development: Compatibility Issues with Mixed Environments

Unicode Programming Summary

To take advantage of the MFC and C run-time support for Unicode, you need to:

- Define **_UNICODE**.
  Define the symbol **_UNICODE** before you build your program.
  Il faut définir UNICODE pour indiquer qu'on est client des fonctions wide et _UNICODE pour indiquer qu'on compile des fonctions wide.
- Specify entry point.
  In the Output page of the Linker folder in the project's Property Pages dialog box, set the Entry Point symbol to **wWinMainCRTStartup**.
  Si la fonction wMain ou wWinMain est définie, alors pas besoin de spécifier le point d'entrée.
- Use "portable" run-time functions and types.
  Use the proper C run-time functions for Unicode string handling. You can use the **wcs** family of functions, but you may prefer the fully "portable" (internationally enabled) **_TCHAR** macros. These macros are all prefixed with **_tcs**; they substitute, one for one, for the **str** family of functions. These functions are described in detail in the Internationalization section of the *Run-Time Library Reference*. For more information, see Generic-Text Mappings in TCHAR.H.
  Use **_TCHAR** and the related portable data types described in Support for Unicode.
- Handle literal strings properly.
  The Visual C++ compiler interprets a literal string coded as
  ```
  L"this is a literal string"
  ```
  to mean a string of Unicode characters. You can use the same prefix for literal characters. Use the **_T** macro to code literal strings generically, so they compile as Unicode strings under Unicode or as ANSI strings (including MBCS) without Unicode. For example, instead of:
  ```
  pWnd->SetWindowText( "Hello" );
  ```
  use:
  ```
  pWnd->SetWindowText( _T("Hello") );
  ```
  With **_UNICODE** defined, **_T** translates the literal string to the L-prefixed form; otherwise, **_T** translates the string without the L prefix.

  Tip The **_T** macro is identical to the **_TEXT** macro.
- Be careful passing string lengths to functions.
  Some functions want the number of characters in a string; others want the number of bytes. For example, if **_UNICODE** is defined, the following call to a **CArchive** object will not work (str is a **CString**):
  ```
  archive.Write( str, str.GetLength( ) );    // invalid
  ```
  In a Unicode application, the length gives you the number of characters but not the correct number of bytes, since each character is two bytes wide. Instead, you must use:
  ```
  archive.Write( str, str.GetLength( ) * sizeof( _TCHAR ) );    // valid
  ```
  which specifies the correct number of bytes to write.
  However, MFC member functions that are character-oriented, rather than byte-oriented, work without this extra coding:
  ```
  pDC->TextOut( str, str.GetLength( ) );
  ```
  CDC::TextOut takes a number of characters, not a number of bytes.

To summarize, MFC and the run-time library provide the following support for Unicode programming under Windows 2000:

- Except for database class member functions, all MFC functions are Unicode-enabled, including **CString**. **CString** also provides Unicode/ANSI conversion functions.
- The run-time library supplies Unicode versions of all string-handling functions. (The run-time library also supplies "portable" versions suitable for Unicode or for MBCS. These are the **_tcs** macros.)
- TCHAR.H supplies portable data types and the **_T** macro for translating literal strings and characters. See Generic-Text Mappings in TCHAR.H.
- The run-time library provides a wide-character version of **main**. Use **wmain** to make your application "Unicode-aware."

Unicode Enabling

To write Unicode-enabled software

1 Use:

- generic data types **TCHAR**, **LPTSTR** for text
- **LPVOID** for pointers of indeterminate type

- explicit types **LPBYTE** for byte pointers
- the TEXT macro
- generic function prototypes

2 Avoid:

- algorithms that assume small character sets
- translation to and from code pages
- assuming a character size

Entry point in DLL and EXE

When you link your image, you either explicitly or implicitly specify an entry point that the operating system will call into after loading the image. For a DLL, the default entry point is **DllMainCRTStartup**. For an EXE, it is **WinMainCRTStartup**.

Linker option

The /ENTRY option specifies a function as the starting address for an .exe file or DLL.

The function must be defined with the **__stdcall** calling convention. The parameters and return value must be defined as documented in the Win32 API for **WinMain** (for an .exe file) or **DllEntryPoint** (for a DLL). It is recommended that you let the linker set the entry point so that the C run-time library is initialized correctly, and C++ constructors for static objects are executed.
By default, the starting address is a function name from the C run-time library. The linker selects it according to the attributes of the program, as shown in the following table.

Function name	Default for
mainCRTStartup (or **wmainCRTStartup**)	An application using /SUBSYSTEM:**CONSOLE**; calls **main** (or **wmain**)
WinMainCRTStartup (or **wWinMainCRTStartup**)	An application using /SUBSYSTEM:**WINDOWS**; calls **WinMain** (or **wWinMain**), which must be defined with **__stdcall**
_DllMainCRTStartup	A DLL; calls **DllMain**, which must be defined with **__stdcall**, if it exists

If the /DLL or /SUBSYSTEM option is not specified, the linker selects a subsystem and entry point depending on whether **main** or **WinMain** is defined.
The functions **main**, **WinMain**, and **DllMain** are the three forms of the user-defined entry point.

Using Generic-Text Mappings

Microsoft Specific

To simplify code development for various international markets, the Microsoft run-time library provides Microsoft-specific "generic-text" mappings for many data types, routines, and other objects. These mappings are defined in TCHAR.H. You can use these name mappings to write generic code that can be compiled for any of the three kinds of character sets: ASCII (SBCS), MBCS, or Unicode, depending on a manifest constant you define using a **#define** statement. Generic-text mappings are Microsoft extensions that are not ANSI compatible.

Preprocessor Directives for Generic-Text Mappings

#define	Compiled version	Example
_UNICODE	Unicode (wide-character)	**_tcsrev** maps to **_wcsrev**
_MBCS	Multibyte-character	**_tcsrev** maps to **_mbsrev**
None (the default: neither **_UNICODE** nor **_MBCS** defined)	SBCS (ASCII)	**_tcsrev** maps to **strrev**

For example, the generic-text function **_tcsrev**, defined in TCHAR.H, maps to **mbsrev** if **MBCS** has been defined in your program, or to **_wcsrev** if **_UNICODE** has been defined. Otherwise **_tcsrev** maps to **strrev**.
The generic-text data type **_TCHAR**, also defined in TCHAR.H, maps to type **char** if **_MBCS** is defined, to type **wchar_t** if **_UNICODE** is defined, and to type **char** if neither constant is defined. Other data type mappings are provided in TCHAR.H for programming convenience, but **_TCHAR** is the type that is most useful.

Generic-Text Data Type Mappings

Generic-text data type name	SBCS (_UNICODE, _MBCS not defined)	_MBCS defined	_UNICODE defined
_TCHAR	**char**	**char**	**wchar_t**
_TINT	**int**	**int**	**wint_t**
_TSCHAR	**signed char**	**signed char**	**wchar_t**
_TUCHAR	**unsigned char**	**unsigned char**	**wchar_t**
_TXCHAR	**char**	**unsigned char**	**wchar_t**
_T or **_TEXT**	No effect (removed by preprocessor)	No effect (removed by preprocessor)	**L** (converts following character or string to its Unicode counterpart)

For a complete list of generic-text mappings of routines, variables, and other objects, see Generic-Text Mappings.
The following code fragments illustrate the use of **_TCHAR** and **_tcsrev** for mapping to the MBCS, Unicode, and SBCS models.

```
_TCHAR *RetVal, *szString;
RetVal = _tcsrev(szString);
```

If **MBCS** has been defined, the preprocessor maps the preceding fragment to the following code:

```
char *RetVal, *szString;
RetVal = _mbsrev(szString);
```

If **_UNICODE** has been defined, the preprocessor maps the same fragment to the following code:

```
wchar_t *RetVal, *szString;
RetVal = _wcsrev(szString);
```

If neither **_MBCS** nor **_UNICODE** has been defined, the preprocessor maps the fragment to single-byte ASCII code, as follows:

```
char *RetVal, *szString;
RetVal = strrev(szString);
```

Thus you can write, maintain, and compile a single source code file to run with routines that are specific to any of the three kinds of character sets.

END Microsoft Specific

String Manipulation

These routines operate on null-terminated single-byte character, wide-character, and multibyte-character strings. Use the buffer-manipulation routines, described in Buffer Manipulation, to work with character arrays that do not end with a null character.

String-Manipulation Routines

Routine	Use
_mbscoll, _mbsicoll, _mbsncoll, _mbsnicoll	Compare two multibyte-character strings using multibyte code page information (**_mbsicoll** and **_mbsnicoll** are case-insensitive)
_mbsdec, _strdec, _wcsdec	Move string pointer back one character
_mbsinc, _strinc, _wcsinc	Advance string pointer by one character
_mbslen	Get number of multibyte characters in multibyte-character string; dependent upon OEM code page
_mbsnbcat	Append, at most, first *n* bytes of one multibyte-character string to another
_mbsnbcmp	Compare first *n* bytes of two multibyte-character strings
_mbsnbcnt	Return number of multibyte-character bytes within supplied character count
_mbsnbcpy	Copy *n* bytes of string
_mbsnbicmp	Compare *n* bytes of two multibyte-character strings, ignoring case
_mbsnbset	Set first *n* bytes of multibyte-character string to specified character
_mbsnccnt	Return number of multibyte characters within supplied byte count
_mbsnextc, _strnextc, _wcsnextc	Find next character in string
_mbsninc, _strninc, _wcsninc	Advance string pointer by *n* characters
_mbsspnp, _strspnp, _wcsspnp	Return pointer to first character in given string that is not in another given string
_mbstrlen	Get number of multibyte characters in multibyte-character string; locale-dependent
_scprintf, _scwprintf	Return the number of characters in a formatted string
_snscanf, _snwscanf	Read formatted data of a specified length from the standard input stream.
sprintf, _stprintf	Write formatted data to a string
strcat, wcscat, _mbscat	Append one string to another
strchr, wcschr, _mbschr	Find first occurrence of specified character in string
strcmp, wcscmp, _mbscmp	Compare two strings
strcoll, wcscoll, _stricoll, _wcsicoll, _strncoll, _wcsncoll, _strnicoll, _wcsnicoll	Compare two strings using current locale code page information (**_stricoll**, **_wcsicoll**, **_strnicoll**, and **_wcsnicoll** are case-insensitive)
strcpy, wcscpy, _mbscpy	Copy one string to another
strcspn, wcscspn, _mbscspn,	Find first occurrence of character from specified character set in string
_strdup, _wcsdup, _mbsdup	Duplicate string
strerror, _wcserror	Map error number to message string
_strerror, __wcserror	Map user-defined error message to string
strftime, wcsftime	Format date-and-time string
_stricmp, _wcsicmp, _mbsicmp	Compare two strings without regard to case
strlen, wcslen, _mbslen, _mbstrlen	Find length of string
_strlwr, _wcslwr, _mbslwr	Convert string to lowercase
strncat, wcsncat, _mbsncat	Append characters of string
strncmp, wcsncmp, _mbsncmp	Compare characters of two strings
strncpy, wcsncpy, _mbsncpy	Copy characters of one string to another
_strnicmp, _wcsnicmp, _mbsnicmp	Compare characters of two strings without regard to case
_strnset, _wcsnset, _mbsnset	Set first *n* characters of string to specified character
strpbrk, wcspbrk, _mbspbrk	Find first occurrence of character from one string in another string
strrchr, wcsrchr, _mbsrchr	Find last occurrence of given character in string
_strrev, _wcsrev, _mbsrev	Reverse string
_strset, _wcsset, _mbsset	Set all characters of string to specified character
strspn, wcsspn, _mbsspn	Find first substring from one string in another string
strstr, wcsstr, _mbsstr	Find first occurrence of specified string in another string
strtok, wcstok, _mbstok	Find next token in string
_strupr, _wcsupr, _mbsupr	Convert string to uppercase
strxfrm, wcsxfrm	Transform string into collated form based on locale-specific information
vsprintf, _vstprint	Write formatted output using a pointer to a list of arguments

_bstr_t

A **_bstr_t** object encapsulates the **BSTR** data type. The class manages resource allocation and deallocation through function calls to **SysAllocString** and **SysFreeString** and other **BSTR** APIs when appropriate. The **_bstr_t** class uses reference counting to avoid excessive overhead.

Construction

```
_bstr_t( ) throw( );
_bstr_t(
   const _bstr_t& s1
) throw( );
_bstr_t(
   const char* s2
) throw( _com_error );
_bstr_t(
   const wchar_t* s3
) throw( _com_error );
_bstr_t(
   const _variant_t& var
) throw ( _com_error );
_bstr_t(
   BSTR bstr,
   bool fCopy
) throw ( _com_error );
```

Remarks

- **_bstr_t( )** Constructs a default **_bstr_t** object that encapsulates a **NULL BSTR** object.
- **_bstr_t(_bstr_t&** *s1* **)** Constructs a **_bstr_t** object as a copy of another. This is a "shallow" copy, which increments the reference count of the encapsulated **BSTR** object instead of creating a new one.
- **_bstr_t(char*** *s2* **)** Constructs a **_bstr_t** object by calling **SysAllocString** to create a new **BSTR** object and encapsulate it. This constructor first performs a multibyte to Unicode conversion.
  If s2 is too large, you may generate a stack overflow error. In such a situation, convert your char* to a wchar_t with MultiByteToWideChar and then call the wchar_t * constructor.
- **_bstr_t(wchar_t*** *s3* **)** Constructs a **_bstr_t** object by calling **SysAllocString** to create a new **BSTR** object and encapsulates it.
- **_bstr_t(_variant_t&** *var* **)** Constructs a **_bstr_t** object from a _variant_t object by first retrieving a **BSTR** object from the encapsulated **VARIANT** object.
- **_bstr_t(BSTR** *bstr* **|** **bool** *fCopy* **)** Constructs a **_bstr_t** object from an existing **BSTR** (as opposed to a **wchar_t*** string). If *fCopy* is **false**, the supplied **BSTR** is attached to the new object without making a new copy with **SysAllocString**. This is the method used by the wrapper functions in the type library headers to encapsulate and take ownership of a **BSTR**, returned by an interface method, in a **_bstr_t** object.

Assign

Copies a **BSTR** into the **BSTR** wrapped by a **_bstr_t**.

```
void Assign(
   BSTR s
) throw( _com_error );
```

Remarks

Assign does a binary copy, which means the entire length of the **BSTR** is copied, regardless of content.

Example

```
// _bstr_t_Assign.cpp
#include <comdef.h>
#include <stdio.h>
int main()
{
   _bstr_t bstrWrapper;   // creates a _bstr_t wrapper
   bstrWrapper = "some text";   // creates BSTR and attaches to it
   wprintf(L"bstrWrapper = %s\n", static_cast<wchar_t*>(bstrWrapper));
```

```
   BSTR bstr = bstrWrapper.Detach();   // bstrWrapper releases its BSTR
   wprintf(L"bstrWrapper = %s\n", static_cast<wchar_t*>(bstrWrapper));
   wprintf(L"bstr = %s\n", bstr);   // "some text"

   bstrWrapper.Attach(SysAllocString(OLESTR("SysAllocedString")));
   wprintf(L"bstrWrapper = %s\n", static_cast<wchar_t*>(bstrWrapper));

   bstrWrapper.Assign(bstr);   // assign a BSTR to our _bstr_t
   wprintf(L"bstrWrapper = %s\n", static_cast<wchar_t*>(bstrWrapper));

   SysFreeString(bstr);   // done with BSTR, do manual cleanup

   bstr= SysAllocString(OLESTR("Yet another string"));   // resuse bstr
   _bstr_t bstrWrapper2 = bstrWrapper;   // two wrappers, one BSTR

   *bstrWrapper.GetAddress() = bstr;
   bstr = 0;   // bstrWrapper and bstrWrapper2 do still point to BSTR
   wprintf(L"bstrWrapper = %s\n", static_cast<wchar_t*>(bstrWrapper));
   wprintf(L"bstrWrapper2 = %s\n", static_cast<wchar_t*>(bstrWrapper2));

   _snwprintf(bstrWrapper.GetBSTR(), bstrWrapper.length(), L"changing BSTR");   // new
value into BSTR
   wprintf(L"bstrWrapper = %s\n", static_cast<wchar_t*>(bstrWrapper));
   wprintf(L"bstrWrapper2 = %s\n", static_cast<wchar_t*>(bstrWrapper2));
}
```

Output

```
bstrWrapper = some text
bstrWrapper = (null)
bstr = some text
bstrWrapper = SysAllocedString
bstrWrapper = some text
bstrWrapper = Yet another string
bstrWrapper2 = some text
bstrWrapper = changing BSTR
bstrWrapper2 = some text
```

Files and Streams

You can open a file by calling the library function fopen with two arguments. The first argument is a filename. The second argument is a C string that specifies:

- Whether you intend to read data from the file or write data to it or both.
- Whether you intend to generate new contents for the file (or create a file if it did not previously exist) or leave the existing contents in place.
- Whether writes to a file can alter existing contents or should only append bytes at the end of the file.
- Whether you want to manipulate a text stream or a binary stream.

Once the file is successfully opened, you can then determine whether the stream is byte oriented (a byte stream) or wide oriented (a wide stream). A stream is initially unbound. Calling certain functions to operate on the stream makes it byte oriented, while certain other functions make it wide oriented. Once established, a stream maintains its orientation until it is closed by a call to fclose or freopen.

	Byte stream A byte stream treats a file as a sequence of bytes. Within the program, the stream looks like the same sequence of bytes, except for the possible alterations described for text/binary stream. Byte mode activated by 1st use of : fgetc, fgets, fread, fscanf, getc, getchar, gets, scanf, ungetc fprintf, fputc, fputs, fwrite, printf, putc, putchar, puts, vfprintf, vprintf	**Wide stream** Within the program, the stream looks like the corresponding sequence of wide characters, except for the possible alterations described for text/binary stream. Wide mode activated by 1st use of : fgetwc, fgetws, fwscanf, getwc, getwchar, ungetwc, wscanf, fwprintf, fputwc, fputws, putwc, putwchar, vfwprintf, vwprintf, wprintf
Text stream To match differing conventions among target environments for representing text in files, the library functions can alter the number and representations of characters transmitted between the program and a text stream		A wide stream treats a text stream as a sequence of generalized multibyte characters, which can have a broad range of encoding rules. Two kinds of character conversions take place: **Unicode-to-MBCS or MBCS-to-Unicode conversion.** When a Unicode stream-I/O function operates in text mode, the source or destination stream is assumed to be a sequence of multibyte characters. Therefore, the Unicode stream-input functions convert multibyte characters to wide characters (as if by a call to the **mbtowc** function). For the same reason, the Unicode stream-output functions convert wide characters to multibyte characters (as if by a call to the **wctomb** function). Conversions between the two representations occur within the Standard C Library. The conversion rules can, in principle, be altered by a call to setlocale that alters the category LC_CTYPE. Each wide stream determines its conversion rules at the time it becomes wide oriented, and retains these rules even if the category LC_CTYPE subsequently changes. **Carriage return – linefeed (CR-LF) translation**. This translation occurs before the MBCS – Unicode conversion (for Unicode stream input functions) and after the Unicode – MBCS conversion (for Unicode stream output functions). During input, each carriage return – linefeed combination is translated into a single linefeed character. During output, each linefeed character is translated into a carriage return – linefeed combination.
Binary stream The library functions do not alter the bytes you transmit between the program and a binary stream. They can, however, append an arbitrary number of null bytes to the file that you write with a binary stream.		The file is assumed to be Unicode, and no CR-LF translation or character conversion occurs during input or output.

Text and Binary Streams

A text stream consists of one or more lines of text that can be written to a text-oriented display so that they can be read. When reading from a text stream, the program reads an NL (newline) at the end of each line. When writing to a text stream, the program writes an NL to signal the end of a line. To match differing conventions among target environments for representing text in files, the library functions can alter the number and representations of characters transmitted between the program and a text stream.

Thus, positioning within a text stream is limited. You can obtain the current file-position indicator by calling fgetpos or ftell. You can position a text stream at a position obtained this way, or at the beginning or end of the stream, by calling fsetpos or fseek. Any other change of position might well be not supported.

For maximum portability, the program should not write:

- Empty files.
- Space characters at the end of a line.
- Partial lines (by omitting the `NL` at the end of a file).
- characters other than the printable characters, NL, and `HT` (horizontal tab).

If you follow these rules, the sequence of characters you read from a text stream (either as byte or multibyte characters) will match the sequence of characters you wrote to the text stream when you created the file. Otherwise, the library functions can remove a file you create if the file is empty when you close it. Or they can alter or delete characters you write to the file.

A binary stream consists of one or more bytes of arbitrary information. You can write the value stored in an arbitrary object to a (byte-oriented) binary stream and read exactly what was stored in the object when you wrote it. The library functions do not alter the bytes you transmit between the program and a binary stream. They can, however, append an arbitrary number of null bytes to the file that you write with a binary stream. The program must deal with these additional null bytes at the end of any binary stream.
Thus, positioning within a binary stream is well defined, except for positioning relative to the end of the stream. You can obtain and alter the current file-position indicator the same as for a text stream. Moreover, the offsets used by ftell and fseek count bytes from the beginning of the stream (which is byte zero), so integer arithmetic on these offsets yields predictable results.

Byte and Wide Streams

A byte stream treats a file as a sequence of bytes. Within the program, the stream looks like the same sequence of bytes, except for the possible alterations described above.
By contrast, a wide stream treats a file as a sequence of generalized multibyte characters, which can have a broad range of encoding rules. (Text and binary files are still read and written as previously described.) Within the program, the stream looks like the corresponding sequence of wide characters. Conversions between the two representations occur within the Standard C Library. The conversion rules can, in principle, be altered by a call to setlocale that alters the category LC_CTYPE. **Each wide stream determines its conversion rules at the time it becomes wide oriented, and retains these rules even if the category LC_CTYPE subsequently changes.**
Positioning within a wide stream suffers the same limitations as for text steams. Moreover, the file-position indicator may well have to deal with a state-dependent encoding. Typically, it includes both a byte offset within the stream and an object of type **mbstate_t**. Thus, the only reliable way to obtain a file position within a wide stream is by calling fgetpos, and the only reliable way to restore a position obtained this way is by calling fsetpos.

Controlling Streams

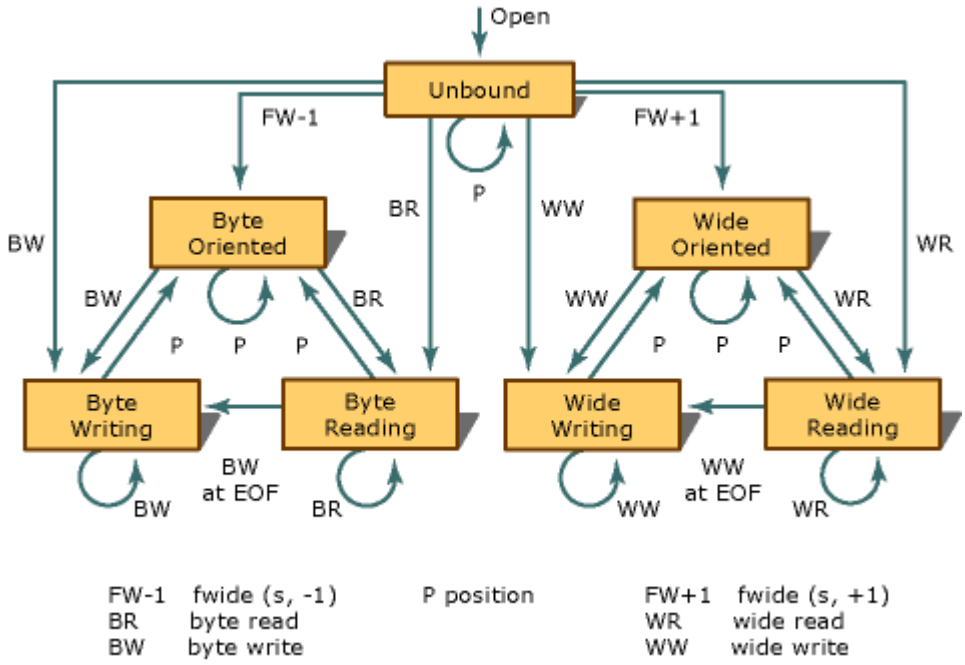
fopen returns the address of an object of type FILE. You use this address as the stream argument to several library functions to perform various operations on an open file. For a **byte stream**, all input takes place as if each character is read by calling fgetc, and all output takes place as if each character is written by calling fputc. For a **wide stream**, all input takes place as if each character is read by calling fgetwc, and all output takes place as if each character is written by calling fputwc.
You can close a file by calling fclose, after which the address of the FILE object is invalid.
A FILE object stores the state of a stream, including:

- An error indicator set nonzero by a function that encounters a read or write error.
- An end-of-file indicator set nonzero by a function that encounters the end of the file while reading.
- A file-position indicator specifies the next byte in the stream to read or write, if the file can support positioning requests.
- A stream state specifies whether the stream will accept reads and/or writes and, with Amendment 1, whether the stream is unbound, byte oriented, or wide oriented.
- A conversion state remembers the state of any partly assembled or generated generalized multibyte character, as well as any shift state for the sequence of bytes in the file).
- A file buffer specifies the address and size of an array object that library functions can use to improve the performance of read and write operations to the stream.

Do not alter any value stored in a FILE object or in a file buffer that you specify for use with that object. You cannot copy a FILE object and portably use the address of the copy as a stream argument to a library function.

Stream States

The valid states, and state transitions, for a stream are shown in the following figure.



Each of the circles denotes a stable state. Each of the lines denotes a transition that can occur as the result of a function call that operates on the stream. Five groups of functions can cause state transitions.
Functions in the first three groups are declared in <stdio.h>:

- The byte read functions — fgetc, fgets, fread, fscanf, getc, getchar, gets, scanf, and ungetc
- The byte write functions — fprintf, fputc, fputs, fwrite, printf, putc, putchar, puts, vfprintf, and vprintf
- The position functions — fflush, fseek, fsetpos, and rewind

Functions in the remaining two groups are declared in <wchar.h>:

- The wide read functions — fgetwc, fgetws, fwscanf, getwc, getwchar, ungetwc, and wscanf,
- The wide write functions — fwprintf, fputwc, fputws, putwc, putwchar, vfwprintf, vwprintf, and wprintf,

The state diagram shows that you must call one of the position functions between most write and read operations:

- You cannot call a read function if the last operation on the stream was a write.
- You cannot call a write function if the last operation on the stream was a read, unless that read operation set the end-of-file indicator.

Finally, the state diagram shows that a position operation never decreases the number of valid function calls that can follow.

Unicode™ Stream I/O in Text and Binary Modes

When a Unicode stream I/O routine (such as **fwprintf**, **fwscanf**, **fgetwc**, **fputwc**, **fgetws**, or **fputws**) operates on a file that is open in text mode (the default), two kinds of character conversions take place:

- Unicode-to-MBCS or MBCS-to-Unicode conversion. When a Unicode stream-I/O function operates in text mode, the source or destination stream is assumed to be a sequence of multibyte characters. Therefore, the Unicode stream-input functions convert multibyte characters to wide characters (as if by a call to the **mbtowc** function). For the same reason, the Unicode stream-output functions convert wide characters to multibyte characters (as if by a call to the **wctomb** function).
- Carriage return – linefeed (CR-LF) translation. This translation occurs before the MBCS – Unicode conversion (for Unicode stream input functions) and after the Unicode – MBCS conversion (for Unicode stream output functions). During input, each carriage return – linefeed combination is translated into a single linefeed character. During output, each linefeed character is translated into a carriage return – linefeed combination.

However, when a Unicode stream-I/O function operates in binary mode, the file is assumed to be Unicode, and no CR-LF translation or character conversion occurs during input or output. Use the _setmode(_fileno(stdin), _O_BINARY); instruction in order to correctly use wcin on a UNICODE text file.

Quick reference

```
int mbtowc(wchar_t *wchar, const char *mbchar, size_t count);
```

- If mbchar is not NULL and if the object that mbchar points to forms a valid multibyte character, mbtowc returns the length in bytes of the multibyte character.
- If mbchar is NULL or the object that it points to is a wide-character null character (L'\0'), the function returns 0.
- If the object that mbchar points to does not form a valid multibyte character within the first count characters, it returns -1.

- The mbtowc function converts count or fewer bytes pointed to by mbchar, if mbchar is not NULL, to a corresponding wide character.
- mbtowc stores the resulting wide character at wchar, if wchar is not NULL.
- mbtowc does not examine more than MB_CUR_MAX bytes (_CRTIMP extern int __mb_cur_max;).

int wctomb(char *mbchar, wchar_t wchar);

- If wctomb converts the wide character to a multibyte character, it returns the number of bytes (which is never greater than MB_CUR_MAX) in the wide character.
- If wchar is the wide-character null character (L'\0'), wctomb returns 1.
- If the conversion is not possible in the current locale, wctomb returns -1.

wint_t integer in wchar.h

- Type of data object that can hold any wide character or wide end-of-file value.

int fgetc(FILE *stream);

- Equivalent to getc, but implemented only as a function, rather than as a function and a macro.

int getc(FILE *stream);

- Same as fgetc, but implemented as a function and as a macro.

wint_t fgetwc(FILE *stream);

- Wide-character version of fgetc.
- Reads c as a multibyte character or a wide character according to whether stream is opened in text mode or binary mode.

wint_t getwc(FILE *stream);

- Wide-character version of getc.
- Reads a multibyte character or a wide character according to whether stream is opened in text mode or binary mode.

int _fgetchar(void);

- Equivalent to fgetc(stdin).
- Also equivalent to getchar, but implemented only as a function, rather than as a function and a macro.
- Microsoft-specific; not ANSI-compatible.

int getchar(void);

- Same as _fgetchar, but implemented as a function and as a macro.

wint_t _fgetwchar(void);

- Wide-character version of _fgetchar.
- Reads c as a multibyte character or a wide character according to whether stream is opened in text mode or binary mode.
- Microsoft-specific; not ANSI-compatible.

wint_t getwchar(void);

- Wide-character version of getchar.
- Reads a multibyte character or a wide character according to whether stream is opened in text mode or binary mode.

Return Value

- Returns the character read.
- To indicate a read error or end-of-file condition :
  - getc and getchar return EOF,
  - fgetc and _fgetchar return EOF,
  - getwc and getwchar return WEOF.
  - fgetwc and _fgetwchar return WEOF.
- Use ferror or feof to check for an error or for end of file.
- For fgetc and fgetwc, if a read error occurs, the error indicator for the stream is set.

Windows & Unicode

Généralités

If a window class was registered with the Unicode version of **RegisterClass**, the window receives only Unicode messages. To determine whether a window uses the Unicode character set or not, call IsWindowUnicode.

RegisterClass

If you register the window class by using **RegisterClassA**, the application tells the system that the windows of the created class expect messages with text or character parameters to use the ANSI character set; if you register it by using **RegisterClassW**, the application requests that the system pass text parameters of messages as Unicode. The IsWindowUnicode function enables applications to query the nature of each window. For more information on ANSI and Unicode functions, see Conventions for Function Prototypes.

IsWindowUnicode

The **IsWindowUnicode** function determines whether the specified window is a native Unicode window.

The character set of a window is determined by the use of the RegisterClass function. If the window class was registered with the ANSI version of **RegisterClass** (**RegisterClassA**), the character set of the window is ANSI. If the window class was registered with the Unicode version of **RegisterClass** (**RegisterClassW**), the character set of the window is Unicode.
The system does automatic two-way translation (Unicode to ANSI) for window messages. For example, if an ANSI window message is sent to a window that uses the Unicode character set, the system translates that message into a Unicode message before calling the window procedure. The system calls **IsWindowUnicode** to determine whether to translate the message.

CallWindowProc

Windows NT/2000/XP: The **CallWindowProc** function handles Unicode-to-ANSI conversion. You cannot take advantage of this conversion if you call the window procedure directly.
Windows 95/98/Me: CallWindowProcW is supported by the Microsoft® Layer for Unicode (MSLU). Also, the ANSI version is supported, to provide more consistent behavior across all Microsoft Windows® operating systems. To use this, you must add certain files to your application, as outlined in Microsoft Layer for Unicode on Windows 95/98/Me Systems.
Unicode : Implemented as Unicode and ANSI versions on Windows NT, Windows 2000, Windows XP

DefWindowProc

Windows 95/98/Me: DefWindowProcW is supported by the Microsoft® Layer for Unicode (MSLU). To use this, you must add certain files to your application, as outlined in Microsoft Layer for Unicode on Windows 95/98/Me Systems.
Unicode : Implemented as Unicode and ANSI versions on Windows NT, Windows 2000, Windows XP

Subclassing and Automatic Message Translation

Subclassing is a technique that allows an application to intercept and process messages sent or posted to a particular window before the window has a chance to process them. The system automatically translates messages into ANSI or Unicode form, depending on the form of the function that subclassed the window procedure.

The following call to the **SetWindowLongA** function subclasses the current window procedure associated with the window identified by the *hwnd* parameter. The new window procedure, `NewWndProc`, will receive messages with text in ANSI format.

```
OldWndProc = (WNDPROC) SetWindowLongA(hwnd,
    GWL_WNDPROC, (LONG)NewWndProc);
```

When `NewWndProc` has finished processing a message, it uses the **CallWindowProc** function as follows to pass the message to `OldWndProc`.

```
CallWindowProc(OldWndProc, hwnd, uMessage, wParam, lParam);
```

If `OldWndProc` was created with a class style of UNICODE, messages will be translated from the ANSI form received by `NewWndProc` into Unicode.

Similarly, a call to the **SetWindowLongW** function would subclass the current window procedure with a window procedure that expects Unicode text messages. Message translation, if necessary, is performed during the processing of the **CallWindowProc** function.

I18N API

National Language Support NLSAPI functions

National Language Support in Windows NT consists of a set of system tables that applications can access through the NLSAPI. The NLSAPI retrieves the following types of information:

- Locale information, such as date, time, number, or currency format, or localized names of countries/regions, languages, or days of the month and week.
- Character mapping tables that map local character encodings (ANSI or OEM) to Unicode or the reverse.
- Keyboard layout information, which, on Windows keyboard layouts, is software-driven. The same keyboard hardware can be used to generate a variety of different language scripts.
- Character typing information. Does a specific Unicode code point represent a letter, a number, a spacing character, or a punctuation symbol? Is a character uppercase or lowercase? For a particular locale, what is the character's uppercase or lowercase equivalent?
- Sorting information—for example, different locales follow different sorting rules for accented characters or may support more than one sorting algorithm.
- Font information. The system stores information about which fonts support which character encoding(s) or which range(s) of Unicode. APIs exist to map which languages the font will support.

APIs to retrieve locale information	APIs to analyze and manipulate strings	APIs to analyze and manipulate system character encoding tables
GetSystemDefaultLangID		
GetUserDefaultLangID		
GetSystemDefaultLCID		
GetUserDefaultLCID		
SetThreadLocale	CompareString	IsValidCodePage
GetThreadLocale IsValidLocale	LCMapString	EnumSystemCodePages
ConvertDefaultLocale	MultiByteToWideChar	GetConsoleCP
EnumSystemLocales	WideCharToMultiByte	GetConsoleOutputCP
GetLocaleInfo	FoldString	SetConsoleCP
SetLocaleInfo	IsDBCSLeadByte	SetConsoleOutputCP
GetTimeFormat	IsDBCSLeadByteEx	GetACP
GetDateFormat	GetStringTypeEx	GetOEMCP
EnumDateFormats(Ex)	GetStringType[A|W]	GetCPInfo
EnumTimeFormats		GetCPInfoEx
EnumCalendarInfo(Ex)		
GetNumberFormat		
GetCurrencyFormat		

Multilingual API functions

APIs to control keyboard layouts	APIs to handle font information	APIs to handle text layout and data
ActivateKeyboardLayout	ChooseFont	DrawTextEx
GetKeyboardLayout	CreateFontIndirectEx	ExtTextOut
GetKeyboardLayoutList	EnumFontFamilies	GetCharacterPlacement
GetKeyboardLayoutName	EnumFontFamiliesEx	GetTextAlign
LoadKeyboardLayout	EnumFontFamExProc	SetTextAlign
MapVirtualKeyEx	GetFontLanguageInfo	GetClipboardData
ToAsciiEx	GetTextCharsetInfo	SetClipboardData
ToUnicodeEx	GetTextFace	GetTextExtent
VkKeyScanEx	TranslateCharsetInfo	
SystemParametersInfo		

Font Technology

To select the appropriate font and output text in local script

1. Use:
   - **EnumFontFamilies** or **ChooseFont** to select fonts
   - **GetTextCharSetInfo** to generate the font signature
   - **GetLocaleInfo** to generate the locale signature
2. Record the charset in your document files
3. Avoid:
   - using **OEM_CHARSET**
   - using **ANSI_CHARSET** by default
   - assuming a given font facename exists

The **Font** common dialog box simplifies the process of creating and selecting fonts. By initializing the **CHOOSEFONT** structure and calling the **ChooseFont** function, an application can support the same font-selection interface that previously required many lines of custom code. (For more information about the **Font** common dialog box, see Common Dialog Box Library.)

Selection by the User

Most font creation and selection operations involve the user. For example, word processing applications let the user select unique fonts for headings, footnotes, and body text. After the user selects a font by using the **Font** dialog box and presses the **OK** button, the **ChooseFont** function initializes the members of a **LOGFONT** structure with the attributes of the requested font. To use this font for text-output operations, an application must first create a logical font and then select that font into its device context. A *logical font* is an application-supplied description of an ideal font. A developer can create a logical font by calling the **CreateFont** or the **CreateFontIndirect** functions. In this case, the application would call **CreateFontIndirect** and supply a pointer to the **LOGFONT** structure initialized by **ChooseFont**. In general, it is more efficient to call **CreateFontIndirect** because **CreateFont** requires several parameters while **CreateFontIndirect** requires only one—a pointer to **LOGFONT**.

Before an application can actually begin drawing text with a logical font, it must find the closest match from the fonts stored internally on the device and the fonts whose resources have been loaded into the operating system. The fonts stored on the device or in the operating system are called *physical fonts*. The process of finding the physical font that most closely matches a specified logical font is called font mapping. This process occurs when an application calls the **SelectObject** function and supplies a handle identifying a logical font. Font mapping is performed by using an internal algorithm that compares the attributes of the requested logical font against the attributes of available physical fonts. When the font mapper algorithm completes its search and determines the closest possible match, the **SelectObject** function returns and the application can begin drawing text with the new font.

The **SetMapperFlags** function specifies whether or not the font mapper algorithm searches only for physical fonts with aspect ratios that match the physical device. The aspect ratio for a device is the ratio formed by the width and the height of a pixel on that device.

The system font (also known as the shell or default font) is the font used for text in the title bars, menus, and dialog boxes. On Windows 95/98/Me and Windows NT, it is MS Sans Serif. On Windows 2000/XP, it is Tahoma.

Special Font Selection Considerations

Although most font selection operations involve the user, there are some instances where this is not true. For example, a developer may want to use a unique font in an application to draw text in a control window. To select an appropriate font, the application must be able to determine what fonts are available, create a logical font that describes one of these available fonts, and then select that font into the appropriate device context.

An application can enumerate the available fonts by using the **EnumFonts** or **EnumFontFamilies** functions. **EnumFontFamilies** is recommended because it enumerates all the styles associated with a family name. This can be useful for fonts with many or unusual styles and for fonts that cross international borders.

Once an application has enumerated the available fonts and located an appropriate match, it should use the values returned by the font enumeration function to initialize the members of a **LOGFONT** structure. Then it can call the **CreateFontIndirect** function, passing to it a pointer to the initialized **LOGFONT** structure. If the **CreateFontIndirect** function is successful, the application can then select the logical font by calling the **SelectObject** function. When initializing the members of the **LOGFONT** structure, be sure to specify a specific character set in the **lfCharSet** member. This member is important in the font mapping process and the results will be inconsistent if this member is not initialized correctly. If you specify a typeface name in the **lfFaceName** member of the **LOGFONT** structure, make sure that the **lfCharSet** value matches the character set of the typeface specified in **lfFaceName**. For example, if you want to select a font such as MS Mincho, **lfCharSet** must be set to the predefined value SHIFTJIS_CHARSET.

The fonts for many East Asian languages have two typeface names: an English name and a localized name. **CreateFont**, **CreateFontIndirect**, and **CreateFontIndirectEx** take the localized typeface name for a system locale

that matches the language, but they take the English typeface name for all other system locales. The best method is to try one name and, on failure, try the other. Note that **EnumFonts**, **EnumFontFamilies**, and **EnumFontFamiliesEx** return the English typeface name if the system locale does not match the language of the font. Starting with Windows 2000, this is no longer a problem because the font mapper for **CreateFont**, **CreateFontIndirect**, and **CreateFontIndirectEx** recognizes either typeface name, regardless of locale.

Bidirectionality

To support bidirectionality in your software

1. Use **GetFontLanguageInfo** and **GetCharacterPlacement** to reorder text:
2. Use **ExtTextOut**

Bidirectional layout

Another assumption is that a character always displays to the right of the characters that precede it in the text. Notice in the example above, we moved the x position to the *right* after each character was input, using these lines:

```
   // Get the next character position.
   GetCharWidth (hDc, (UINT) wParam, (UINT) wParam, &cCharWidths) ;
   // BAD! Don't do this!
   g_xStartChar += cCharWidths ;
```

Correctly determining the position of the next character in the stream would require implementing the Unicode algorithm for layout of bidirectional text (BiDi algorithm), which is a major undertaking indeed. Instead, use **ExtTextOut** on the whole buffer, as shown above, and let the system implementation of the BiDi algorithm handle layout.

However, there may be other cases where your application assumes left to right (LTR) layout, such as the x position passed in the call to **ExtTextOut**. You can make this selectable by the user, and set the proper x value as follows:

```
Static UINT uiAlign = TA_LEFT ;
int nxStartBuffer ;

case WM_PAINT :

   hDc = BeginPaint (hWnd, &ps) ;

   SelectObject (hDc, hTextFont) ;

   // Set the x position for right or left aligned text.
   if (uiAlign & TA_RIGHT) {   // Start at right edge.
    nxStartBuffer = rcRectLine.right ;
   } else {         // Start at left edge.
    nxStartBuffer = XSTART ;
   }

   SetTextAlign (hDc, uiAlign) ;

   // Same as above.
   ExtTextOut (hDc, nxStartBuffer, nyStartBuffer, ETO_OPAQUE,
   &rcRectLine, szOutputBuffer, nChars, NULL) ;
```

Vertical Writing

To implement vertical writing in your software

1. Use fonts with @ in front of facename
2. Set escapement and orientation to 270°
3. Check your algorithms for:
    - coordinates calculation
    - caret positioning
    - caret orientation
    - virtual key handling

Changing Input Language

	To handle changing the input language properly
1	Add code to manage **WM_INPUTLANGCHANGEREQUEST** and **WM_INPUTLANGCHANGE**
2	Use **GetLocaleInfo** and either **TranslateCharsetInfo** or **GetTextCharsetInfo** to determine if language is supported by available fonts
3	Use **ActivateKeyboardLayout** to activate a specific layout

Text Output

Text-Formatting Attributes

An application can use six functions to set the text-formatting attributes for a device context: **SetBkColor**, **SetBkMode**, **SetTextAlign**, **SetTextCharacterExtra**, **SetTextColor**, and **SetTextJustification**. These functions affect the text alignment, the intercharacter spacing, the text justification, and text and background colors. In addition, six other functions can be used to retrieve the current text formatting attributes for any device context: **GetBkColor**, **GetBkMode**, **GetTextAlign**, **GetTextCharacterExtra**, **GetTextColor**, and **GetTextExtentPoint32**.

Text Alignment

Applications can use the **SetTextAlign** function to specify how the system should position the characters in a string of text when they call one of the drawing functions. This function can be used to position headings, page numbers, callouts, and so on. The system positions a string of text by aligning a reference point on an imaginary rectangle that surrounds the string, with the current cursor position or with a point passed as an argument to one of the text drawing functions. The **SetTextAlign** function lets the application specify the location of this reference point. The following is a list of the possible reference point locations.

Location	Description
left/bottom	The reference point is located at the bottom-left corner of the rectangle.
left/base line	The reference point is located at the intersection of the character-cell base line and the left edge of the rectangle.
left/top	The reference point is located at the top-left corner of the rectangle.
center/bottom	The reference point is located at the center of the bottom of the rectangle.
center/base line	The reference point is located at the intersection of the character-cell base line and the center of the rectangle.
center/top	The reference point is located at the center of the top of the rectangle.
right/bottom	The reference point is located at the bottom-right corner of the rectangle.
right/base line	The reference point is located at the intersection of the character-cell base line and the right edge of the rectangle.
right/top	The reference point is located at the top-right corner of the rectangle.

The default text alignment for a device context is the upper-left corner of the imaginary rectangle that surrounds the text. An application can retrieve the current text-alignment setting for any device context by calling the **GetTextAlign** function.

Intercharacter Spacing

Applications can use the **SetTextCharacterExtra** function to alter the intercharacter spacing for all text output operations in a specified device context.
The default intercharacter spacing value for any device context is zero. An application can retrieve the current intercharacter spacing value for a device context by calling the **GetTextCharacterExtra** function.

Text Justification

Applications can use the **GetTextExtentPoint32** and **SetTextJustification** functions to justify a line of text. Text justification is a common operation in any desktop publishing and in most word processing applications. The **GetTextExtentPoint32** function computes the width and height of a string of text. After the width is computed, the application can call the **SetTextJustification** function to distribute extra spacing between each of the words in a line of text.

Text and Background Color

Applications can use the **SetTextColor** function to set the color of text drawn in the client-area of their windows, as well as the color of text drawn on a color printer. An application can use the **SetBkColor** function to set the color

that appears behind each character and the **SetBkMode** function to specify how the system should blend the selected background color with the current color or colors on the video display.
The default text color for a display device context is black; the default background color is white; and the default background mode is OPAQUE. An application can retrieve the current text color for a device context by calling the **GetTextColor** function. An application can retrieve the current background color for a device context by calling the **GetBkColor** function and the current background mode by calling the **GetBkMode** function.

Character Widths

Applications need to retrieve character-width data when they perform such tasks as fitting strings of text to page or column widths. There are four functions that an application can use to retrieve character-width data. Two of these functions retrieve the character-advance width and two of these functions retrieve actual character-width data.
An application can use the **GetCharWidth32** and **GetCharWidthFloat** functions to retrieve the advance width for individual characters or symbols in a string of text. The advance width is the distance that the cursor on a video display or the print-head on a printer must advance before printing the next character in a string of text. The **GetCharWidth32** function returns the advance width as an integer value. If greater precision is required, an application can use the **GetCharWidthFloat** function to retrieve fractional advance-width values.
An application can retrieve actual character-width data by using the **GetCharABCWidths** and **GetCharABCWidthsFloat** functions. The **GetCharABCWidthsFloat** function works with all fonts. The **GetCharABCWidths** function only works with TrueType and OpenType fonts. For more information about TrueType and OpenType fonts, see Raster, Vector, TrueType, and OpenType Fonts.

String Widths and Heights

In addition to retrieving character-width data for individual characters, applications also need to compute the width and height of entire strings. Two functions retrieve string-width and height measurements: **GetTextExtentPoint32**, and **GetTabbedTextExtent**. If the string does not contain tab characters, an application can use the **GetTextExtentPoint32** function to retrieve the width and height of a specified string. If the string contains tab characters, an application should call the **GetTabbedTextExtent** function.
Applications ca use the **GetTextExtentExPoint** function for word-wrapping operations. This function returns the number of characters from a specified string that fit within a specified space.

Font Ascenders and Descenders

Some applications determine the line spacing between text lines of different sizes by using a font's maximum ascender and descender. An application can retrieve these values by calling the **GetTextMetrics** function and then checking the **tmAscent** and **tmDescent** members of the **TEXTMETRIC**.
The maximum ascent and descent are different from the typographic ascent and descent. In TrueType and OpenType fonts, the typographic ascent and descent are typically the top of the f glyph and bottom of the g glyph. An application can retrieve the typographic ascender and descender for a TrueType or OpenType font by calling the **GetOutlineTextMetrics** function and checking the values in the **otmMacAscent** and **otmMacDescent** members of the **OUTLINETEXTMETRIC** structure.

Font Dimensions

An application can retrieve the physical dimensions of a TrueType or OpenType font by calling the **GetOutlineTextMetrics** function. An application can retrieve the physical dimensions of any other font by calling the **GetTextMetrics** function. To determine the dimensions of an output device, an application can call the **GetDeviceCaps** function. **GetDeviceCaps** returns both physical and logical dimensions.
A logical inch is a measure the system uses to present legible fonts on the screen and is approximately 30 to 40 percent larger than a physical inch. The use of logical inches precludes an exact match between the output of the screen and printer. Developers should be aware that the text on a screen is not simply a scaled version of the text that will appear on the page, particularly if graphics are incorporated into the text.

Drawing Text

After an application selects the appropriate font, sets the required text-formatting options, and computes the necessary character width and height values for a string of text, it can begin drawing characters and symbols by calling any of the text-output functions:

- **DrawText**
- **DrawTextEx**
- **ExtTextOut**
- **PolyTextOut**
- **TabbedTextOut**

- **TextOut**

When an application calls one of these functions, the operating system passes the call to the graphics engine, which in turn passes the call to the appropriate device driver. At the device driver level, all of these calls are supported by one or more calls to the driver's own **ExtTextOut** or **TextOut** function. An application will achieve the fastest execution by calling **ExtTextOut**, which is quickly converted into an **ExtTextOut** call for the device. However, there are instances when an application should call one of the other three functions; for example, to draw multiple lines of text within the borders of a specified rectangular region, it is more efficient to call **DrawText**. To create a multicolumn table with justified columns of text, it is more efficient to call **TabbedTextOut**

Complex Scripts

While the functions discussed in the preceding work well for many languages, they may not deal with the needs of complex scripts. *Complex scripts* are languages whose printed form is not rendered in a simple way. For example, a complex script may allow bidirectional rendering, contextual shaping of glyphs, or combining characters. Due to these special requirements, the control of text output must be very flexible.

Windows 2000/XP: Functions that display text—**TextOut**, **ExtTextOut**, **TabbedTextOut**, **DrawText**, and **GetTextExtentExPoint**—have been extended to support complex scripts. In general, this support is transparent to the application. However, applications should save characters in a buffer and display a whole line of text at one time, so that the complex script shaping modules can use context to reorder and generate glyphs correctly. In addition, because the width of a glyph can vary by context, applications should use **GetTextExtentExPoint** to determine line length rather than using cached character widths.

In addition, complex script-aware applications should consider adding support for right-to-left reading order and right alignment to their applications. You can toggle the reading order or alignment between left and right with the following code:

```
// Save lAlign (this example uses static variables)
static LONG lAlign = TA_LEFT;

// When user toggles alignment (assuming TA_CENTER is not supported).

lAlign = TA_RIGHT;

// When the user toggles reading order.

lAlign = TA_RTLREADING;

// Before calling ExtTextOut, for example, when processing WM_PAINT

SetTextAlign (hDc, lAlign);
```

To toggle both attributes at once, execute the following statement and then call **SetTextAlign** and **ExtTextOut**, as shown previously:

```
lAlign = TA_RIGHT^TA_RTLREADING;
```

You can also process complex scripts with Uniscribe. Uniscribe is a set of functions that allow a fine degree of control for complex scripts. For more information, see Uniscribe and Processing Complex Scripts.

Context-sensitive characters

The second assumption you need to discard is that a given character in a given font always looks the same, and has the same properties. Characters in languages such as Arabic change shape depending on the surrounding characters. Specifically, Arabic characters take one of four forms—initial, medial, final, and stand-alone—depending on where they occur in the text stream. Moreover, adjacent Arabic characters often ligate, meaning they combine together in a single glyph called a ligature.

This means you cannot use the old trick of putting out characters one by one, as you get them in the **wParam** parameter from the **WM_CHAR** message. If you do, then the system cannot do the contextual shaping for you, because when it comes time to render a character, the system does not know what characters precede or follow. It also means that you should not cache character widths and compute line lengths yourself, since the width of the character depends on the context. For example, this code will produce incorrect results when displaying most complex scripts:

```
case WM_CHAR:

   // NOTE: This is an example of what *not* to do, because
   // characters that should join or otherwise interact
   // typographically will show as separate, stand-alone characters.
   hDc = GetDC (hWnd) ;   // BTW, this is also bad for other reasons!
```

```
   SelectObject (hDc, hTextFont) ;
   SetBkMode (hDc, TRANSPARENT) ;
   ExtTextOut (hDc, g_xStartOneChar, YSTART, 0,
    NULL, (LPCTSTR) &wParam, 1, NULL) ;

   // Get the next character position.
   GetCharWidth (hDc, (UINT) wParam, (UINT) wParam, &cCharWidths) ;
   // This assumes left to right scripts, so it will break on
   // Arabic and Hebrew!
   g_xStartChar += cCharWidths ;

   ReleaseDC (hWnd, hDc) ;
   Return 0 ;
```

Instead, you should save characters in a buffer, and put out the entire buffer each time a new character is typed, as follows:

```
RECT rcRectLine ;

case WM_CHAR:

szOutputBuffer[nChars] = (TCHAR) wParam ;
   if (nChars < BUFFER_SIZE-1) {   // Limited by the buffer size.
    nChars++ ;
   }
   // This will generate a WM_PAINT message, where all of
   // the text buffer is displayed at once. This is the
   // recommended approach.
   InvalidateRect (hWnd, &rcRectLine, TRUE) ;
   Return 0 ;

case WM_PAINT :

   hDc = BeginPaint (hWnd, &ps) ;

   SelectObject (hDc, hTextFont) ;

   // Write the whole text buffer in the line buffer rectangle.
   // This happens every time the user enters a character.
   ExtTextOut (hDc, nxStartBuffer, nyStartBuffer, ETO_OPAQUE,
      &rcRectLine, szOutputBuffer, nChars, NULL) ;

   EndPaint (hWnd, &ps) ;

   return 0 ;
```

Window Layout

Window Layout and Mirroring

The window layout defines how text and Microsoft® Windows® Graphics Device Interface (GDI) objects are laid out in a window or device context (DC). Some languages, such as English, French, and German, require a left-to-right (LTR) layout. Other languages, such as Arabic and Hebrew, require right-to-left (RTL) layout. The window layout applies to text but also affects the other GDI elements of the window, including bitmaps, icons, the location of the origin, buttons, cascading tree controls, and whether the horizontal coordinate increases as you go left or right. For example, after an application has set RTL layout, the origin is positioned at the right edge of the window or device, and the number representing the horizontal coordinate increases as you move left. However, not all objects are affected by the layout of a window. For example, the layout for dialog boxes, message boxes, and device contexts that are not associated with a window, such as metafile and printer DCs, must be handled separately. Specifics for these are mentioned later in this topic.

The window functions allow you to specify or change the window layout in Arabic and Hebrew versions of Windows 98 and Windows Millennium Edition (Windows Me), and in all versions of Windows 2000 or later. Note that changing to a RTL layout (also known as mirroring) is not supported for windows that have the style CS_OWNDC or for a DC with the GM_ADVANCED graphic mode.

By default, the window layout is left-to-right (LTR). To set the RTL window layout, call **CreateWindowEx** with the style WS_EX_LAYOUTRTL. Also by default, a child window (that is, one created with the WS_CHILD style and with a valid parent *hWnd* parameter in the call to CreateWindow or **CreateWindowEx**) has the same layout as its parent. To disable inheritance of mirroring to all child windows, specify WS_EX_NOINHERITLAYOUT in the call to **CreateWindowEx**. Note, mirroring is not inherited by owned windows (those created without the WS_CHILD style) or those created with the parent *hWnd* parameter in **CreateWindowEx** set to NULL. To disable inheritance of mirroring for an individual window, process the WM_NCCREATE message with GetWindowLong and **SetWindowLong** to turn off the WS_EX_LAYOUTRTL flag. This processing is in addition to whatever other processing is needed. The following code fragment shows how this is done.

```
SetWindowLong (hWnd,
               GWL_EXSTYLE,
               GetWindowLong(hWnd,GWL_EXSTYLE) & ~WS_EX_LAYOUTRTL))
```

You can set the default layout to RTL by calling SetProcessDefaultLayout(LAYOUT_RTL). All windows created after the call will be mirrored, but existing windows are not affected. To turn off default mirroring, call **SetProcessDefaultLayout**(0).

Note, **SetProcessDefaultLayout** mirrors the DCs only of mirrored windows. To mirror any DC, call SetLayout(hdc, LAYOUT_RTL). For more information, see the discussion on mirroring device contexts not associated with windows, which comes later in this topic.

Bitmaps and icons in a mirrored window are also mirrored by default. However, not all of these should be mirrored. For example, those with text, a business logo, or an analog clock should not be mirrored. To disable mirroring of bitmaps, call **SetLayout** with the LAYOUT_BITMAPORIENTATIONPRESERVED bit set in *dwLayout*. To disable mirroring in a DC, call **SetLayout**(hdc, 0).

To query the current default layout, call GetProcessDefaultLayout. Upon a successful return, *pdwDefaultLayout* contains LAYOUT_RTL or 0. To query the layout settings of the device context, call GetLayout. Upon a successful return, **GetLayout** returns a **DWORD** that indicates the layout settings by the settings of the LAYOUT_RTL and the LAYOUT_BITMAPORIENTATIONPRESERVED bits.

After a window has been created, you change the layout using the **SetWindowLong** function. For example, this is necessary when the user changes the user interface language of an existing window from Arabic or Hebrew to German. However, when changing the layout of an existing window, you must invalidate and update the window to ensure that the contents of the window are all drawn on the same layout. The following snippet is from sample code that changes the window layout as needed:

Show Example

```
// Using ANSI versions of GetWindowLong and SetWindowLong because Unicode
// is not needed for these calls

lExStyles = GetWindowLongA(hWnd, GWL_EXSTYLE);

// Check whether new layout is opposite the current layout
if (!!(pLState -> IsRTLLayout) != !!(lExStyles & WS_EX_LAYOUTRTL))
{
   // the following lines will update the window layout

   lExStyles ^= WS_EX_LAYOUTRTL;       // toggle layout
   SetWindowLongA(hWnd, GWL_EXSTYLE, lExStyles);
```

```
   InvalidateRect(hWnd, NULL, TRUE);    // to update layout in the client area
}
```
In mirroring, you should think in terms of "near" and "far" instead of "left" and "right". Failure to do so can cause problems. One common coding practice that causes problems in a mirrored window occurs when mapping between screen coordinates and client coordinates. For example, applications often use code similar to the following to position a control in a window:
```
// DO NOT USE THIS IF APPLICATION MIRRORS THE WINDOW

// get coordinates of the window in screen coordinates
GetWindowRect(hControl, (LPRECT) &rControlRect);

// map screen coordinates to client coordinates in dialog
ScreenToClient(hDialog, (LPPOINT) &rControlRect.left);
ScreenToClient(hDialog, (LPPOINT) &rControlRect.right);
```
This causes problems in mirroring because the left edge of the rectangle becomes the right edge in a mirrored window, and vice versa. To avoid this problem, replace the **ScreenToClient** calls with a call to **MapWindowPoints** as follows:
```
// USE THIS FOR MIRRORING

GetWindowRect(hControl, (LPRECT) &rControlRect);
MapWindowPoints(NULL, hDialog, (LPPOINT) &rControlRect, 2)
```
This code works because, on platforms that support mirroring, **MapWindowPoints** is modified to swap the left and right point coordinates when the client window is mirrored. For more information, see the Remarks section of MapWindowPoints.
Another common practice that can cause problems in mirrored windows is positioning objects in a client window using offsets in screen coordinates instead of client coordinates. For example, the following code uses the difference in screen coordinates as the x position in client coordinates to position a control in a dialog box.
Show Example
```
// OK if LTR layout and mapping mode of client is MM_TEXT,
// but WRONG for a mirrored dialog

RECT rdDialog;
RECT rcControl;

HWND hControl = GetDlgItem(hDlg, IDD_CONTROL);
GetWindowRect(hDlg, &rcDialog);            // gets rect in screen coordinates
GetWindowRect(hControl, &rcControl);
MoveWindow(hControl,
        rcControl.left - rcDialog.left,   // uses x position in client coords
        rcControl.top - rcDialog.top,
        nWidth,
        nHeight,
        FALSE);
```
This code is fine when the dialog window has left-to-right (LTR) layout and the mapping mode of the client is MM_TEXT, because the new x position in client coordinates corresponds to the difference in left edges of the control and the dialog in screen coordinates. However, in a mirrored dialog, left and right are reversed, so instead you should use **MapWindowPoints** as follows:
```
RECT rcDialog;
RECT rcControl;

HWND hControl - GetDlgItem(hDlg, IDD_CONTROL);
GetWindowRect(hControl, &rcControl);

// MapWindowPoints works correctly in both mirrored and non-mirrored windows.
MapWindowPoints(NULL, hDlg, (LPPOINT) &rcControl, 2);

// Now rcControl is in client coordinates.
MoveWindow(hControl, rcControl.left, rcControl.top, nWidth, nHeight, FALSE)
```

Mirroring Dialog Boxes and Message Boxes

Dialog boxes and message boxes do not inherit layout, so you must set the layout explicitly. To mirror a message box, call MessageBox or MessageBoxEx with the MB_RTLREADING option. To layout a dialog box right-to-left, use the extended style WS_EX_LAYOUTRTL in the dialog template structure DLGTEMPLATEEX. Property sheets are a special case of dialog boxes. Each tab is treated as a separate dialog box, so you need to include the WS_EX_LAYOUTRTL style in every tab that you want mirrored.

Mirroring Device Contexts Not Associated with a Window

DCs that are not associated with a window, such as metafile or printer DCs, do not inherit layout, so you must set the layout explicitly. To change the device context layout, use the **SetLayout** function.

The **SetLayout** function is rarely used with windows. Typically, windows receive an associated DC only in processing a WM_PAINT message. Occasionally, a program creates a DC for a window by calling GetDC. Either way, the initial layout for the DC is set by BeginPaint or **GetDC** according to the window's WS_EX_LAYOUTRTL flag.

The values returned by GetWindowOrgEx, GetWindowExtEx, GetViewportOrgEx and GetViewportExtEx are not affected by calling **SetLayout**.

When the layout is RTL, GetMapMode will return MM_ISOTROPIC instead of MM_TEXT. Calling SetMapMode with MM_TEXT will function correctly; only the return value from **GetMapMode** is affected. Similarly, calling **SetLayout**(hdc, LAYOUT_RTL) when the mapping mode is MM_TEXT causes the reported mapping mode to change to MM_ISOTROPIC.

Uniscribe

Uniscribe is a set of APIs that allow a fine degree of control for processing complex scripts. A complex script requires special processing to display and edit because the characters, or glyphs, are not laid out in a simple way. The rules governing the shaping and positioning of glyphs are specified and catalogued in *The Unicode Standard: Worldwide Character Encoding, Version 2.0,* Addison-Wesley Publishing Company.

About Complex Scripts

A complex script has at least one of the following attributes:

- Allows bidirectional rendering.
- Has contextual shaping.
- Has combining characters.
- Has specialized word-breaking and justification rules.
- Filters out illegal character combinations.

Bidirectional rendering refers to the script's ability to handle text that reads both left-to-right and right-to-left. For example, in the bidirectional rendering of Arabic, the default reading direction for text is right-to-left, but for some numbers, it is left-to-right. Processing a complex script must account for the difference between the logical (keystroke) order and the visual order of the glyphs. In addition, processing must properly deal with caret movement and hit testing. The mapping between screen position and a character index for, say, selection of text or caret display requires knowledge of the layout algorithms.
Contextual shaping occurs when a script's characters change shape depending on the characters that surround them. This occurs in English cursive writing when a lowercase "l" changes shape depending on the character that precedes it such as an "a" (connects low to the "l") or an "o" (connects high). Arabic is a script that exhibits contextual shaping.
Combining characters or ligatures are characters that join into one character when placed together. One example is the "ae" combination in English; it is sometimes represented by a single character. Arabic is a script that has many combining characters.
Specialized word break and justification refers to scripts that have complex rules for dividing words between lines or justifying text on a line. Thai is such a script.
Filtering out illegal character combinations occurs when a language does not allow certain character combinations. Thai is such a script.

Processing Complex Scripts

The following are options for processing complex scripts:

- Text functions
- Edit controls
- Rich edit controls
- Uniscribe

The options you choose will depend on the following factors:

- The type of text or scripts.
- The choice of implementation model that is used, for example, the text layout and whether the application handles line breaking.
- Whether the application exists or is created from scratch.

In general, an application that does relatively simple script processing can choose any option. However, for the most complete control of complex script processing, Uniscribe is recommended.

Text Functions

Applications that work mostly in plain text--that is, text that uses the same typeface, weight, color, and so on--have traditionally written text and measured line lengths using standard text functions, such as **TextOut**, **ExtTextOut**, **TabbedTextOut**, **DrawText**, and **GetTextExtentExPoint**. Starting with Microsoft® Windows® 2000, these functions have been extended to support complex scripts. In general, this support is transparent to the application. However, applications should save characters in a buffer and display the whole line of text at one time rather than, for example, calling **ExtTextOut** on each character as it is typed in by the user. This allows the complex script shaping modules to use context to reorder and generate glyphs correctly. Also, applications should use **GetTextExtentExPoint** to determine line length rather than computing line lengths from cached character widths. This is because the width of a glyph may vary by context. In addition, complex script-aware applications should consider adding support for right-to-left reading order and right alignment to their applications.
For more information, see Fonts and Text.

Edit Controls

The standard edit controls have been extended to support multilingual text and complex scripts. This includes not only input and display, but also correct cursor movement over character clusters (in Thai and Devanagari script, for example).
For more information, see Edit Controls.

Rich Edit Controls

Rich Edit 3.0 is a higher-level collection of interfaces that takes advantage of Uniscribe to further insulate text layout clients from the complexities of certain scripts. Rich Edit is designed for clients whose primary purpose is not necessarily text layout, but who nonetheless need to display complex scripts.
Rich Edit provides fast, versatile editing of rich Unicode multilingual text and simple plain text. It includes extensive message and COM interfaces, text editing, formatting, line breaking, simple table layout, vertical text layout, bidirectional text layout, Indic and Thai support, an editing UI much like Microsoft Word, and Text Object Model interfaces. Rich Edit is the simplest way for a client to support features of complex scripts. Clients use its **TextOut** function to automatically parse, shape, position, and break lines.
For more information, see Rich Edit Controls.

Uniscribe

Uniscribe enables extremely fine processing of complex scripts. It supports the complex rules found in scripts such as Arabic, Indian, and Thai. It also handles scripts written from right to left, such as Arabic and Hebrew, and supports the mixing of scripts.