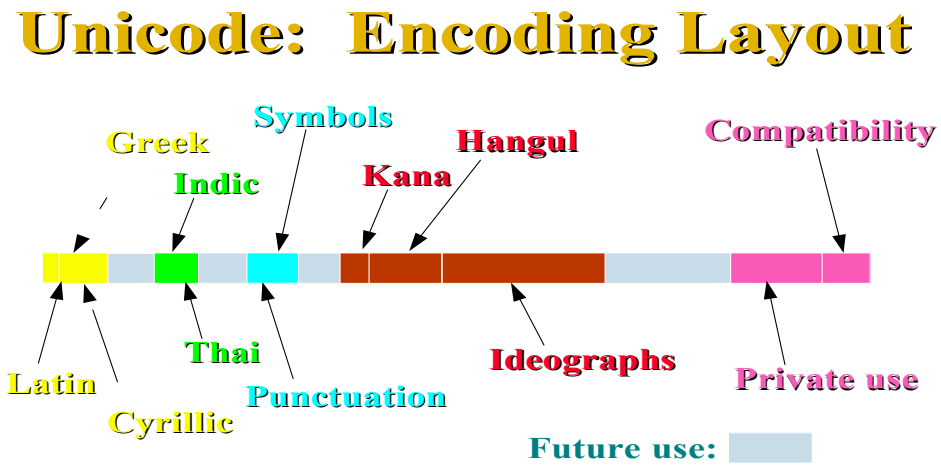
Unicode™ Support in Win32®

Unicode™ is a 16-bit, fixed width character encoding standard that encompasses virtually all of the characters commonly used on computers today. This includes most of the world's written languages, plus publishing characters, mathematical and technical symbols, and punctuation marks.

The Unicode Consortium was founded in 1991 as a non-profit organization dedicated to devising and promoting the Unicode Standard. Its membership now includes companies such as Adobe, Aldus, Borland, Digital, GO, IBM, HP, Lotus, Metaphor, Microsoft, NeXT, Novell, Sun, Symantec, Taligent, Unisys and WordPerfect.

Addison Wesley has published a book called *The Unicode Standard Version 1.0* (Vol. 1: ISBN 0-201-56788-1, Vol. 2: ISBN 0-201-60845-6). Unicode Version 1.1 is code-for-code identical to the first page of the international standard ISO10646. Information on The Unicode Standard, Version 1.1 can be found on the MSDN Developer Library in the Specs and Strategy, Specifications section.



Windows 3.1 and ANSI

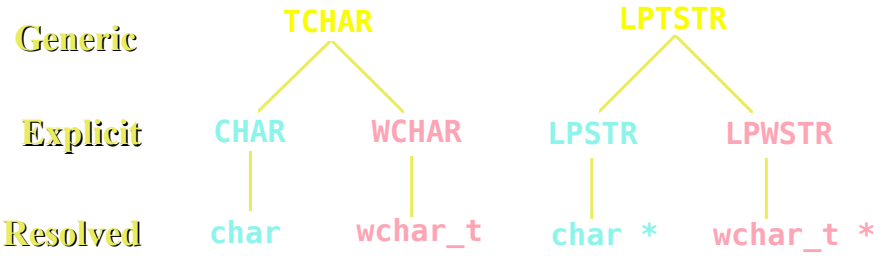
Unlike the Windows NT™ operating system, which is based on Unicode, the Windows™ operating system, version 3.1, uses the code page model. Each single-byte code page is limited to 256 different characters. The U.S. and Western European versions of Windows 3.1 use code page 1252. Eastern European versions use code page 1250 if they are Latin-based, and code page 1251 if they are Cyrillic. There are also code pages for Greek, Turkish, Thai, Arabic, and Hebrew. The code-page model adds a layer of complexity in the Far East versions of Windows 3.1 (Japanese, Korean, traditional and simplified Chinese). Each of these systems is based on a double-byte character set (DBCS), where some characters are represented by one-byte values and the rest are represented by two-byte values. For this reason, the term "multi-byte character set," which means one or more bytes, is sometimes used in place of "DBCS."

All of the Windows 3.1 code pages support the set of characters used in English, but suppose you wanted to support text from different code page categories (Swedish, Arabic, Chinese, and Turkish) in your application documents. With the code page model, this is not very practical. In addition, what if you needed to support a language for which no code page exists? Unicode is a fixed-width, 16-bit

character encoding that covers the majority of written languages used in the world today. Each Unicode code point is associated with one and only one character. While it isn't a localization panacea, it sets the stage for seamless support of a much broader range of languages than the code page model.

Unicode and the Win32 API

The Win32® API is designed so that each system function exists in two flavors: one that expects string parameters to be in a Windows 3.1-based (single-byte ANSI) character encoding, and another that expects string parameters to be in Unicode. Only a single name for each function appears in the documentation, but in the system there are two different entry points. The function prototype (e.g. **SetWindowText()**) in the header files is a macro that expands depending on whether the compile time symbol UNICODE is defined (usually by adding -DUNICODE to the compiler's command line). Unicode programs that call any C run-time functions should also define the _UNICODE flag (UNICODE preceded by an underscore). The compiled name appends either an A (for ANSI, e.g. **SetWindowTextA()**) or a W (for wide character, or Unicode, e.g. **SetWindowTextW()**) to the function names. Similarly, the header files define generic data types (TCHAR, LPTSTR), and data structures. With these, it is possible to use a single set of sources and compile them for either Unicode or ANSI support.



To convert data (e.g. from a file) from a Windows 3.1-based character encoding to Unicode or vice-versa, you can use the two functions **MultiByteToWideChar()** and **WideCharToMultiByte()**.

The Win32 API is supported on several platforms. These platforms differ in their level of support for Unicode.

Unicode in Windows NT

Windows NT uses Unicode internally. Windows NT GDI does all its text work in Unicode, resource strings are compiled as Unicode, system information files are stored as Unicode, and the Windows NT file system (NTFS) file names are Unicode. However, even on Windows NT an application can be written to use one of the Windows (ANSI) or MS-DOS® (OEM) code pages. An 'A' version of a Win32 API entry point calls **MultiByteToWideChar()** to translate text parameters, and then calls the 'W' version of the same API. Conversely, the system calls **WideCharToMultiByte()** to translate return parameters for a program that expects ANSI values.

Thus, you could write an application for the Japanese version of Windows NT that uses the Shift-JIS (CP 932) character encoding internally. Shift-JIS is the ANSI code page supported by Japanese Windows NT. In all language versions of Windows NT, ANSI and MS-DOS code pages are also supported for MS-DOS or Win16 applications that run on Windows NT.

Some Windows NT-based applications may call APIs that require an ANSI or OEM code page number as a parameter. The macros CP_ACP (for ANSI code page) or CP_OEMCP (for OEM code page) will resolve to the default code page values that have been set by the user.

Win32-based programs that are written to the Unicode (or 'W') version of the API run natively across all language versions of Windows NT — no character translation needs to be performed.

Unicode in Win32s

Win32s™ API offers a strategy for targeting both the Windows 3.1 and the Windows NT operating systems with one 32-bit binary. Win32s does not natively support all of the wide character versions of the Win32 API entry points. Depending on where an application is installed, the target display mechanism (GDI) might be Windows 3.1, and the file system might be (FAT); neither of these supports Unicode. Win32s does, however, support APIs that convert between Unicode and the native Windows 3.1 code page. Win32s version 1.2 also supports a number of functions that are useful in 'processing' Unicode data, such as **CompareStringW()**, **LCMapstringW()**, etc. This makes it possible for a Windows 3.1-based application to share a Unicode-based file format with its Windows NT-based sister application, even though Windows 3.1 does not use Unicode internally.

Unicode in Windows "Chicago"

Windows "Chicago" inherits the Unicode support present in Win32s and adds support for the CF_UNICODETEXT clipboard format. Chicago does not internally support the full set of Unicode string processing APIs available on Windows NT.

However, a new feature in Chicago, called multilingual content I/O, is a method of dynamically switching among single-byte character sets (the same character sets used in Windows 3.1) for displaying fonts and determining keyboard layouts. Using formatted text, i.e. text that stores font tags, it is possible with Chicago to create and display a single document that spans multiple character sets, such as Cyrillic, Greek, Eastern European, and Western European. Such text can no longer be expressed as plain text, unless it is converted into Unicode. Unicode thus becomes an attractive plain-text format for exporting multilingual data on Chicago.

Unicode in OLE and Windows "Cairo"

Windows NT "Daytona" supports OLE 2.0 and ships with its 32-bit libraries. Windows NT "Cairo" will support additional OLE features, including full object support for distributed applications and distributed file systems. Given the reality of global computer networks, these technologies must offer consistent, reliable, language-independent access to data. Local character sets are too limited — they don't interoperate with one another adequately. Therefore, Windows NT "Cairo" and 32-bit OLE 2.0 (even on Chicago) are based on Unicode, and full access to all Cairo-specific features will require Unicode.

Tools

Historically, most compiler and resource compiler source files have been in ASCII (7-bit) or ANSI (8-bit) format. This is adequate for the Americas and parts of Europe, assuming those applications will never have to manipulate data from other regions. More recent compilers support the Japanese Shift-JIS code page. To use Shift-JIS resources elsewhere in the Far East or with other Japanese code pages you will have to do some editing.

Development tools which support universal, language-independent applications on Win32 are becoming available. For example, Microsoft Visual C++™ development system version 2.0 supports the Win32 Unicode APIs, and its Microsoft Foundation Classes 3.0 (MFC) provide support for ANSI, DBCS (Shift-JIS) and Unicode text processing. The resource and message compilers that will ship with Windows "Daytona," Windows "Chicago" and Visual C++ 2.0 will support Unicode files.

Implications for your strategy

Win32-based applications running on Windows NT will run more efficiently if they are based on Unicode. This is especially true for programs localized into Japanese or other Far East languages. Japanese-language programs that use the Shift-JIS character encoding must contend with the mix of one- and two-byte characters. Since Unicode is fixed-width, string parsing is simpler. And since the Windows NT system converts non-Unicode APIs to Unicode APIs at runtime, not using Unicode adds

a step (overhead) to some function calls. Another benefit to using wide characters internally is that only the NLS (National Language Support) data table for Unicode needs to be present — multiple data tables to support different code pages are unnecessary.

Standalone Win32-based applications running on Chicago will operate more efficiently at runtime if they're based on ANSI or another local code page. Documents from those applications can use Chicago's multilingual content I/O if more than one single-byte character set is required. However, in a client/server world you need to consider how closely your product will interact with operating sytems offering full Unicode support. The Unicode conversion APIs on Windows Chicago offer a path to Unicode-based data that may reside on a server, and at a minimum your strategy should include the use of these conversion APIs. If your application requires working with language-independent data, you may want to incorporate Unicode conventions within your application's internal processing modules, then convert data before sending it to the system. Windows NT and Windows NT "Cairo" put Unicode "on the wire," unless they are communicating with "downlevel" (e.g. Windows for Workgroups or LAN Manager) workstations.

Conclusion

Unicode is the preferred method for server applications. You should not assume that servers will only deal with data originating the Americas or parts of Europe. On all Win32-based systems (including Chicago), Unicode conventions are used in 32-bit OLE 2.0. Unicode can make localization to the Far East simpler, since text handling in your code can be implemented with consistent, language-independent algorithms instead. There is no need for extensive code reworking for specific languages, as you need to do for the different DBCS encodings. And Unicode brings us closer to Information At Your Fingertips by offering a language-independent encoding for distributed applications and data on a global network.

Additional Reading

- The Unicode Standard 1.0 (plus 1.1 supplement), published by Addison-Wesley
- Microsoft Win32 Programmer's Reference (overview on Unicode)
- Microsoft International Handbook for Software Design

Appendix A: Code Page Data
(excerpted from the Microsoft Win32 NLS API Specification)

The following table shows various Code Pages and the support associated with that code page. Some code pages can be used in the console or file system (OEMCP), or in the Windows UI (ACP). Other code pages are supported only for some functions, such as code page translation to/from Unicode (Macintosh Code Pages, EBCDIC). Except for in Chicago, which has multilingual content I/O support, only one ACP is active at a time in a system. Unicode is not designated as ACP or OEMCP in this table, but it may be used as the character encoding for GUI and console Win32 applications on Windows NT, Daytona and Cairo.

CP ID - the Code Page ID
Name - the canonical name of the character set
ACP - eligibility for ACP
OEMCP - eligibility for OEMCP

CP ID	Name	ACP	OEMCP	US/Eur. NT 3.1	Chicago	Daytona	Cairo
1200	Unicode (BMP of ISO 10646)			x		x	x
1250	Windows 3.1 Eastern European	x		x	x	x	x
1251	Windows 3.1 Cyrillic	x		x	x	x	x
1252	Windows 3.1 US (ANSI)	x		x	x	x	x
1253	Windows 3.1 Greek	x		x	x	x	x
1254	Windows 3.1 Turkish	x		x	x	x	x
1255	Hebrew	x			x	x	x
1256	Arabic	x			x	x	x
1257	Baltic	x			x	x	x
437	MS-DOS United States		x	x	x	x	x
708	Arabic (ASMO 708)		x		x	x	x
709	Arabic (ASMO 449+, BCON V4)		x		x	x	x
710	Arabic (Transparent Arabic)		x		x	x	x
720	Arabic (Transparent ASMO)		x		x	x	x
737	Greek (formerly 437G)		x	x	x	x	x
775	Baltic		x		x	x	x
850	MS-DOS Multilingual (Latin I)		x	x	x	x	x
852	MS-DOS Slavic (Latin II)		x	x	x	x	x
855	IBM Cyrillic		x	x	x	x	x
857	IBM Turkish		x	x	x	x	x
860	MS-DOS Portuguese		x	x	x	x	x
861	MS-DOS Icelandic		x	x	x	x	x
862	Hebrew		x		x	x	x
863	MS-DOS Canadian-French		x	x	x	x	x
864	Arabic		x		x	x	x
865	MS-DOS Nordic		x	x	x	x	x
866	MS-DOS Russian (USSR)		x	x	x	x	x
869	IBM Modern Greek		x	x	x	x	x
874	Thai	x	x		x	x	x
932	Japan	x	x	NT-J	x	NT-J	C-J
936	Simplified Chinese (PRC, Singapore)	x	x		x	x	x
949	Korean	x	x		x	x	x
950	Chinese (Taiwan, Hong Kong)	x	x		x	x	x
874	Thai		x		x	x	x
1361	Korean (Johab)		x				
10000	Macintosh Roman			x		x	x
10001	Macintosh Japanese				x	x	x
10006	Macintosh Greek I			x		x	x

10007	Macintosh Cyrillic			x		x	x
10029	Macintosh Latin 2			x		x	x
10079	Macintosh Icelandic				x	x	x
10081	Macintosh Turkish			x		x	x
037	EBCDIC			x		x	x
500	EBCDIC "500V1"			x		x	x
1026	EBCDIC			x		x	x
875	EBCDIC			x		x	x

Appendix B: NLSAPI Support (excerpted from the Microsoft NLSAPI Specification)

Code Pages:
There are a total of four code page settings in Windows NT and Chicago, and a large overall number of code pages supported as options. The ANSI Code Page (ACP) is supported for Windows 3.1 compatibility. The Console Code Page, Console Output Code Page and OEM Code Page (OEMCP) are supported for MS-DOS compatibility. There is a high-level relationship between the locale that the system is running in and the various code page settings. Based on the locale specified during installation, default code page settings are derived for these 4 values. Only the Console and Console Output Code Pages can be modified without re-installing Windows NT. Other code pages are available, based on the installed locale, for use in data translation. These include secondary OEM code pages, MAC code pages and EBCDIC code pages.

Unicode:
In order to support Windows 3.1 and MS-DOS compatibility, Windows NT supports 8-bit code pages via the previous four code page settings. All character sets and code pages supported by Microsoft systems and applications can be mapped to Unicode (wide characters). Chicago supports Unicode only within data translation. The Chicago operating system is based on the Windows ANSI code page, rather than Unicode.

ANSI defines the "wchar_t" data type to refer to a wide character. We use the convention **WCHAR**, **LPWCHAR** and **LPWSTR** to represent a wide character, pointer to a wide character, and a pointer to a wide character string respectively.

APIs:
The API set described in this table are broken down into three areas: string transformation, code page manipulation and locale manipulation. The types of string transformation that are supported are uppercasing, lowercasing, sort key generation (all locale dependent), getting string type information and character translation from one code page to Unicode and back again, i.e., round-trip mapping, (both non-locale dependent). Code Page manipulation includes getting and setting the 2 Console Code Pages and getting information about the other 2 code pages being used. Locale manipulation includes comparing strings, mapping strings, and getting information about installed locales.

The Windows NT API names use a trailing W to denote the use of the wide character set (Unicode) and the Chicago APIs use an A to denote the use of the ANSI character set.

API Name	Win32S v1.2	NT 3.1	Chicago	Daytona	Cairo
GetSystemDefaultLangID	X	X	X	X	X
GetUserDefaultLangID	X	X	X	X	X
GetSystemDefaultLCID	X	X	X	X	X
GetUserDefaultLCID	X	X	X	X	X
IsValidLocale	X	X	X	X	X
ConvertDefaultLocale	X		X	X	X
EnumSystemLocales	X			X	X
EnumSystemLocalesW	X			X	X
EnumSystemLocalesA	X		X	X	X
GetLocaleInfo	X			X	X
GetLocaleInfoW	X	X		X	X
GetLocaleInfoA	X		X	X	X
SetLocaleInfo	TBD			X	X
SetLocaleInfoW	TBD	X		X	X
SetLocaleInfoA	TBD			X	X
GetTimeFormat	TBD			X	X
GetTimeFormatW	TBD	X		X	X
GetTimeFormatA	X		X	X	X
GetDateFormat	TBD			X	X
GetDateFormatW	TBD	X		X	X
GetDateFormatA	X		X	X	X
EnumDateFormats	TBD			X	X
EnumDateFormatsW	TBD			X	X

EnumDateFormatsA	X		X	X	X
EnumTimeFormats	TBD			X	X
EnumTimeFormatsW	TBD			X	X
EnumTimeFormatsA	X		X	X	X
EnumCalendarInfo	TBD			X	X
EnumCalendarInfoW	TBD			X	X
EnumCalendarInfoA	X		X	X	X
GetNumberFormat	TBD			X	X
GetNumberFormatW	TBD			X	X
GetNumberFormatA	X		X	X	X
GetCurrencyFormat	TBD			X	X
GetCurrencyFormatW	TBD			X	X
GetCurrencyFormatA	X		X	X	X
CompareString	X			X	X
CompareStringW	X	X		X	X
CompareStringA	X		X	X	X
SetThreadLocale	TBD	X		X	X
GetThreadLocale	X	X		X	X
LCMapString	X			X	X
LCMapStringW	X	X		X	X
LCMapStringA	X		X	X	X
MultiByteToWideChar	X	X	X	X	X
WideCharToMultiByte	X	X	X	X	X
FoldString	TBD			X	X
FoldStringW	TBD	X		X	X
FoldStringA	TBD			X	X
IsValidCodePage	X	X	X	X	X
EnumSystemCodePages	TBD			X	X
EnumSystemCodePagesW	TBD			X	X
EnumSystemCodePagesA	TBD		X	X	X
GetConsoleCP		X		X	X
GetConsoleOutputCP		X		X	X
SetConsoleCP		X		X	X
SetConsoleOutputCP		X		X	X
GetACP	X	X	X	X	X
GetOEMCP	X	X	X	X	X
GetCPInfo	X	X	X	X	X
IsDBCSLeadByte	X	X	X	X	X
IsDBCSLeadByteEx			X		X
GetStringTypeEx			X	X	X
GetStringTypeW	X	X		X	X
GetStringTypeA	X		X	X	X
GetStringTypeExA			X	X	X
GetStringTypeExW				X	X