P E O P L E S O F T , I N C .

UNICODE IN PEOPLETOOLS 8:

CASE STUDY OF A LARGE-SCALE HETEROGENEOUS UNICODE IMPLEMENTATION

TOBY PHIPPS
PEOPLETOOLS PRODUCT MANAGER - GLOBALIZATION
PEOPLESOFT, INC.

Abstract

PeopleSoft 8 will be the first major enterprise resource planning system to implement Unicode in a generally available release. This project was primarily undertaken in PeopleTools, the underlying technology behind all PeopleSoft Applications, and involved significant changes to well over 5 million lines of code, yet was completed in less than 5 months. Unlike most Unicode conversion projects involving development tools where not only the underlying technology needs to be changed but also the higher-level application definitions built using these tools, the goal of the project was to move to Unicode with minimal disruption to the existing application definitions and business rules built using the toolset.

Given the extensive breadth of the PeopleSoft applications suite, the move to Unicode not only had to encompass changes to the core C and C++ code in the toolset, but also had to cope with:

- 7 database management system backends
- 8 back-end operating system platforms, including IBM EBCDIC platforms
- 15 third-party integrated products
- Unicode Executable support on Win95/Win98

This paper examines some of the techniques used in the project to accomplish to Unicode integration while other development was in progress, how operating system specific issues were solved, and how the Unicode story was internally marketed to the applications development organization and upper management. Additionally, it will briefly discuss the architecture of the PeopleSoft 8 release, and how the Unicode standard plays an important role in all tiers of the system.

COPYRIGHT ©2000 PEOPLESOFT, INC.

Introduction to PeopleSoft Application Architecture

The PeopleSoft suite of applications encompasses well over 50 separate application products addressing business requirements in areas including Enterprise Resource Planning, Human Resource Management, Enterprise Performance Management, Higher Education and Customer Relationship Management. PeopleSoft solutions address specific business needs, are adaptable to changing requirements, and provide easy information access. At the heart is PeopleTools, our innovative technology platform and toolset. Packaged with all PeopleSoft applications, PeopleTools provides a single foundation for developing, deploying, and maintaining enterprise-wide solutions. PeopleTools is used by PeopleSoft Application Development to build the business applications delivered by PeopleSoft, but is also delivered to our customers, enabling a wide range of users--from systems analysts to functional users--to quickly adapt PeopleSoft applications to meet specific business needs.

A key feature of PeopleTools based applications is that they are independent of the database management system, hardware platform or operating system of their ultimate host system. Applications built in PeopleTools can be deployed on any of the PeopleSoft supported hardware, operating system and database platforms without modification. PeopleTools applications are developed using a native Microsoft Windows Win32 development toolset,and a pure Internet HTML-based client.

Built in C++, PeopleTools insulates application developers, administrators and end users from the complexities of each back-end database and operating system and provides a workbench for building complex applications using an object-oriented, component approach. It provides an interpretive script language, PeopleCode for coding business logic and other application rules best expressed programmatically.

The Migration to Unicode - an Overview

Until Release 8, all PeopleTools components relied on native ANSI character sets for all transient and persistent data storage and management. The back-end SQL database provided management of character data, and was also responsible for any character set conversion that may be required between the application database and the PeopleTools clients. Typically the character sets of client and server machines matched, and the conversions were limited to ASCII / EBCDIC transcoding. Although the entire PeopleTools suite has been double-byte enabled for some time, allowing correct processing of Chinese, Korean and Japanese data in several popular multi-byte character sets, several limitations existed in the character set awareness of the PeopleTools clients.

Firstly, PeopleTools had little awareness of non-Latin1 based single-byte character sets. Thus, problems frequently arose with character transformations, case conversion, language-sensitive collation and line breaking when non-Latin1 single-byte character data was managed in a PeopleSoft system. Secondly, as most supported database systems only allowed a single character set per database instance, PeopleSoft applications were extremely limited in the repertoire of characters that could be maintained and managed in a single database. Several large, worldwide PeopleSoft implementations had to implement multiple back-end databases, each using a different ANSI character set in order to allow maintenance of data in the national language of each of their target rollout countries.

As a result, given demands for a larger language support repertoire, PeopleSoft was faced with enabling support for well over 10 new character sets within the PeopleTools suite, as well as for solving the single-database, multiple character set issue. Instead of heading down this multi-character set path, the use of Unicode for in-memory and persistent storage was a logical choice.

Several obstacles stood in the way of implementing the Unicode Standard across all of PeopleTools. Specifically:

- PeopleTools shares a single C++ codebase across 10 different operating system platforms, including Windows NT/2000, Windows 95/98/ME, 4 UNIX platforms and IBM OS/390 (MVS). Any Unicode implementation within this codebase would need to be portable across all these systems off the same codeline.

- Seven database management systems are supported by PeopleTools, including database products from IBM, Microsoft, Oracle, Informix and Sybase. Although not all of these databases could be expected to support an end-to-end Unicode implementation immediately, they all had to be able to co-exist with a single set of PeopleTools Unicode binaries.

- PeopleTools 8 feature development was well underway, with over 100 C++ developers working on the codebase simultaneously. Any mass Unicode conversion in this codebase had to take place with minimal disruption to existing development projects already underway. The codebase had to be in a "compilable" state throughout the conversion, either as an ANSI or UNICODE executable set.

- A single executable client must be supported for Windows 95, Windows 98, Windows NT 4.0 and Windows 2000 to minimize the overhead of supporting dual executable sets. Given the lack of native Unicode support in Windows 95 and 98, this posed a major technical obstacle.

- PeopleTools integrates approximately 15 third party products in the toolset. Not all of these products could be expected to be available as Unicode versions in the timeframe required for Release 8. Products that were not Unicode enabled would need to be integrated with API-boundary ANSI conversions.

- Several thousand application developers use the PeopleTools product line on a daily basis to build complex business applications. Any change to the PeopleTools architecture must have a minimum impact on these developers. Similarly, any applications already developed using previous versions of PeopleTools must continue to function with new toolset unchanged, but still derive the benefits of an end-to-end Unicode solution.

Redefining Base Quantities

Based on the need for transparency in adopting Unicode for our existing application products with minimal impact, the core character types used within PeopleTools (formally represented as bytes) were changed to represent Unicode characters, or more precisely, UTF-16 code units. Not only were the internal C++ datatypes re-defined to be Unicode characters, but also the higher-level application definitions built using PeopleTools were adjusted to represent multiples of characters rather than multiples of bytes. This was done instead of continuing to use byte-based datatypes, and expanding the sizes to cater for UCS-2 or UTF-8 byte-to-character ratios, which would cause significant changes in not only application datatype definitions, but also any string manipulation functions.

Given this wide-reaching impact of the byte-to-character change, it was critically important to select a consistent Unicode transformation for data representation within memory, and for presentation to the development user base.

As this approach allowed existing byte-based application definitions managed by PeopleTools to now manage data in terms of characters, all application character quantities formerly defined in terms of bytes were re-defined to be the same multiple of characters. PeopleCode, PeopleSoft's proprietary scripting language was similarly re-defined to operate at the Unicode character level instead of the byte level. By re-defining the basic quantities being operated on not only by the C++ code, but also by the application definitions manipulated and managed by this code, existing applications could continue to function as designed, but would operate on Unicode data. Physical storage of course would be significantly larger than in previous releases due to characters being stored in Unicode, however this was managed by the lower-level C++ code, and would be transparent to the application developer community.

Choosing a Transformation

A wide variety of Unicode transformations was considered for each of the four main processing areas needing to manage data: the application client, data on-the-wire between client and server, the application server and finally, the back-end database. UTF-8, UCS-2, UCS-4 and UTF-16 were all considered. UCS-2 was selected for in-memory processing on both the client and server tiers, firstly because the standard Microsoft Windows wide character APIs accepted only UCS-2, and secondly because it provided an easy migration path to UTF-16 as a future project by re-using much of the existing double-byte enablement code within the legacy ANSI codebase. Particular attention was also placed on ensuring both processing tiers; the client and the server; used the same transformation, as the C++ code is shared between these two platforms, and significant differences in memory allocation would prove problematic.

As the vast majority of the data transmitted between the server and client machines is still ASCII-based, UTF-8 proved to be a more efficient and compact solution for network encoding due to its 1:1 byte relationship with ASCII. In fact, the use of UTF-8 on the network proved to be more efficient than compressing UCS-2 data at the sender side and uncompressing UCS-2 at the receiver. It had the extra benefit of being byte order independent; allowing each processing tier to have ignorance of each other's byte order. During serialization, all data is converted to the byte-order-neutral UTF-8, and de-serialized into UCS-2 by the receiver into their native byte order.

Finally, database vendors control the Unicode transformation available in their database. As of writing, most of the database systems supported by PeopleSoft supported either UCS-2 or UTF-8 data storage, however no database supported both. Therefore, the transformation used for in-database storage will vary from database system to database system, and PeopleTools converts to UCS-2 at the database API boundary.

The complete Unicode migration involved three distinct components; all related to the PeopleTools development products:

- C++ Codebase Migration. The core PeopleTools toolset is written primarily in C and C++. This code, managed and maintained by approximately 100 developers, is the development tool used by the entire company for building enterprise applications.

- Database Engine Migration. The back-end data store for all PeopleSoft applications is one of seven SQL-based relational databases. Although all support SQL as an access method, each boast different implementations of the SQL standard and have database-specific programmatic interfaces. Similarly, implementation of Unicode character representation varies significantly from vendor-to-vendor. PeopleTools

must be able to support the existing database software, and where possible, allow customers to store and manage data in Unicode at the database level.

- Application Definition Migration. Finally, existing application definitions maintained in PeopleTools repositories need to operate against Unicode data, both in memory and in the database, without the need for manual conversions by application developers.

C++ Codebase Migration

Migrating the C++ codebase to Unicode is the single most disruptive phase of the migration project. The PeopleTools development environment itself is written in C++, and manages not only the definition of the PeopleSoft applications, but also the runtime environment. A clean migration of the PeopleTools suite to Unicode would result in the existing application definitions automatically inheriting the new character-vs.-byte semantics and provide interfaces to both ANSI and Unicode back-end database engines and third party products.

Cross-Platform Unicode Functions

Before modifying any core C or C++ code, cross-platform wide-character compatibility problems had to be addressed to continue to allow a single codebase to successfully execute and compile across the 11 platforms, while sharing common memory allocation and interoperability. Although the ANSI C standard prescribes the use of the **wchar_t** datatype for wide character representation within C programs, the encoding or Unicode transformation used by these datatypes is not standardized, and varies widely across platforms.

A brief survey of our 11 target platforms showed that the Microsoft platforms all supported the use of 16-bit Unicode UCS-2 values for wchar_t, several Unix versions implemented Unicode UCS-4 32-bit values, and others implemented proprietary widened DBCS 16-bit characters. Additionally, a common set of wide character string functions was not available across all the platforms.

This disparity required the construction of a cross-platform C wide-character compatibility library, to provide not only the basic wide-character equivalents of the ANSI C string functions, but also common trans-coding and transformation operations.

Built with the use of a third-party Unicode transcoding and transformation engine, Rosette from Basis Technology Corp., this library provides approximately 75 wide-character functions equivalent to the basic C runtime library string operations, all operating on 16-bit UCS-2 character data. This library effectively replaces the wide-character functions offered by each target operating system, with the exception of the Microsoft Win32 platforms, where the native OS-provided functions are used.

Generic Datatype Conversion

To accomplish the move to a fully Unicode enabled codebase without significant downtime of the continuing ANSI-based development, the initial Unicode datatype conversion used generic datatypes and functions that could be resolved at compile time to either ANSI narrow character datatypes and functions or their wide-character variants. Microsoft's Visual C++ compiler provides a suite of generic datatypes and functions that perform just this function, based on the setting of a compile-time directive, -DUNICODE. The key generic datatype in Microsoft's implementation is TCHAR, which resolves to the byte based CHAR datatype without the -DUNICODE directive, and the 16-bit UCS-2 based WCHAR with the directive.

Along the same lines, each of the standard C runtime library string functions also have generic equivalents, _tcs*functionname* which resolve to str*functionname* or wcs*functionname* (narrow and wide variants respectively), based on the same compile-time flag. These generic datatypes allow a single codebase to be compiled as an ANSI or Unicode executable based on a single compile-time directive.

Finally, inline C constants needed to be converted to wide characters at compile time. Once again, the Microsoft compilers take care of this by converting any string constant proceeded by a 'L' as a Unicode constant at compile-time based on the character set of the compile machine, or an override specified in the source file. Thus, the constant string "Hello World!" will be interpreted as an ANSI constant in the compiler's character set at compile time, and L"Hello World!" will be first converted to a Unicode string before compile. Similarly to the generic datatypes, Microsoft also provides a generic constant marker, "_T" which resolves to "L" for a Unicode build (this indicating a Unicode string constant to the compiler), and resolves to a null string for an ANSI build.

Of course, these generic functions are specific to the Microsoft compilers, so a cross-platform equivalent had to be written with the use of several simple header-file macros. Additionally, the "L" Unicode string constant marker, although supported by all the compilers on each platform, resolves to the local compiler's wide-char variant. Given that PeopleTools uses UCS-2 exclusively for wide character strings, these strings had to always be resolved as UCS-2 strings at compile-time. Therefore, a pre-compiler had to be written for the non-Microsoft compilers to scan for the L"String" constants, and convert these to Unicode constants in an interim file before the final compile. This is no simple task given the multitude of syntax constructs where the L"string" constants are possible!

Once these cross-platform libraries and headers were in-place to ensure that UCS-2 style wide-character handling was available and tested across all the PeopleTools target platforms, a single, massive conversion of all character datatypes and string functions to their generic equivalents took place across the entire codebase. This conversion, consisting of a series of AWK macros converted all 75+ string manipulation functions to their Microsoft-style _tcs*functionname* equivalents, all CHAR datatypes (and their typedef equivalents) to TCHAR, and all string constants to _T"string" generic string constants. Run over a weekend with exclusive access to the master source tree, this conversion affected approximately every second line of code across the entire 5 million line codebase.

Dual-Build Phase

At the completion of the TCHAR generic datatype conversion weekend, PeopleTools could still be compiled and executed successfully without the -DUNICODE compile directive, as the generic character header macros simply mapped the new generic datatypes back to ANSI characters, the new generic string functions back to their C standard ANSI string functions, and the Unicode constants back to ANSI constants. For the majority of developers, there was no impact on their existing ANSI builds. However, from this point forward, the use of ANSI functions and datatypes was prohibited, and build-time audits were developed to weed-out accidental usages. Byte-based datatypes were made available for situations where developers specifically needed to work with byte quantities, where previously a CHAR datatype was used.

Given the massive, blind conversion of all CHAR datatypes to Unicode, it was expected that the newly converted codebase not compile as a Unicode executable. Not all uses of the CHAR datatype were really dealing with characters, but often used also for byte-based operations such as bit fields, flags, and targets for byte-based operations such as encryption. Given the generic datatype architecture, the same codebase could be shared between core developers continuing work on feature development with ANSI executables, and a Unicode conversion team, focusing on preparing the code to be built as a Unicode executable, cleaning up some of the errors made by the automated generic datatype conversion processes.

This clean-up process took several weeks with approximately 5 full-time developers to get to a point where the same codebase could be compiled and run as a Unicode executable set or as an ANSI executable set with similar quality. Much of the work was complicated by the fact that other development continued simultaneously, with developers still unused to working with generic datatypes. Still, many features operated exclusively in ANSI mode, primarily due to developers not conforming to the generic datatype specifications.

Migrating to a Single-Unicode Build

To minimize ongoing quality and dual-build maintenance and testing issues, the development effort shifted focus exclusively to one build, the Unicode build once this build's quality met basic standards. From this date forward, the Unicode build became the exclusive version of the product for all developers and the ANSI build was all but deprecated. By moving the focus onto the Unicode build exclusively, significant initial productivity losses were to be expected, as each developer would need to complete the testing and migration of their module to Unicode before continuing development work. Because of code ownership and complexity issues, this was seen as the best approach, and proved to be successful. Although the core Unicode conversion developers were responsible for the basic stability of the Unicode build, and the initial codebase-wide Unicode changes, each module owner assumed responsibility for the finer points of completing the Unicode conversion and testing for their product component.

The result was a single, functioning Unicode PeopleTools product build approximately 5 weeks after the initial generic datatype conversion introduced Unicode datatypes to well over 5 million lines of code.

Single Executable Client Support for Windows 95 / 98 / NT / 2000

A significant restriction of Unicode executables on Windows operating systems is the severely limited wide character API repertoire available on Windows 95 and 98 platforms.

Microsoft does not support the Unicode API on the Windows 95 and 98 platforms. Currently, Win32 applications that call any Unicode API's, other than TextOutW and a small collection of similar functions, fail on Windows 95 and Windows 98 platforms. MFC applications using Unicode API's will not even initialize on these platforms.

One solution to this problem would be to create two separate executable sets from the single PeopleTools codebase; a Unicode set for execution on Windows NT and 2000 clients, and an ANSI set for Windows 95 and 98. This would pose a significant development and support cost however, as all code would need to be compiled and tested in both modes, similarly, product updates or patches provided would also have to be compiled and provided in each mode.

Instead, PeopleSoft contracted with Basis Technology Corp to implement a proposal originally put forward by Microsoft's F. Avery Bishop at the 14th International Unicode Conference. (http://www.unicode.org/iuc/iuc14/a301.html). This proposal involved building a set of "satellite" DLLs designed to emulate much of the Win32 Wide Character API, and to convert the calls back to ANSI if the application is running on Windows 95 or 98.

Although more complex than originally scoped, this project proved largely successful, allowing a single Unicode set of PeopleTools executables to execute on Windows 95, 98, NT and 2000. The key requirement was to provide this enablement with only minor changes to the link step of the PeopleTools build process; no source changes were possible. When running on Windows 95 or 98, PeopleTools continues to run as a Unicode application, however any operating system interface calls are converted to ANSI. Therefore, full end-to-end Unicode support can still only be truly accomplished using a Windows NT or 2000 client,

however users requiring compatibility only with the local Windows ANSI character set can continue to use Windows 95 or 98 workstations. Further information on Basis Technology Corp's Cheops product can be found at http://www.basistech.com/products/Cheops.html.

Database Engine Migration

The first step in migrating the back-end database engine to Unicode is to determine the support for Unicode across each database vendor's products in three specific areas:

- SQL DDL Statement Constructs
- Database Static Limits
- Database Application Programmatic Interface

However, regardless of the Unicode functionality provided by each database engine (if any), the Unicode-enabled PeopleTools suite must maintain at least the existing support for ANSI data encoding at the database level. As all data in-memory in PeopleTools is now managed in Unicode, this involves Unicode to ANSI conversion at the database API boundary in the case of non-Unicode back-end databases.

SQL DDL Statement Constructs

Unfortunately, the ANSI SQL standard provides little guideline for SQL database vendors in implementing Unicode-based SQL datatypes. The current ANSI standard provides for two character sets to be maintained by a database; a default character set and a *national* character set. Each of these two character sets has a suite of character datatypes for declaring columns containing character data. Typically, these are known as CHAR and NCHAR, and their variants (VARCHAR / NVARCHAR, TEXT / NTEXT etc.)

However, the standard does not prescribe what character set should be used for each of the two character type variants, how this character set is specified, nor how the quantities of characters in each datatype are measured. This has resulted in a series of differing implementations of Unicode character types across database vendors.

Currently, the most popular implementation appears to be to allow the declaration of Unicode (or one of its transformations) as the database-wide character set, to be used for all data stored in CHAR datatypes. Column length specifications typically continue to be specified in terms of bytes.

Thus, in a database using UTF-8 as the database character set, a VARCHAR(30) column can store up to 30 bytes of UTF-8 data, equating to between 10 and 30 Unicode characters, depending on the number of bytes in each character. This leads to uncertainty in terms of character storage limits, as the number of characters allowable in a given column will vary based on the language of the text being stored.

As of writing, Oracle, Informix, Sybase and IBM DB2/UNIX databases take this approach to Unicode storage declaration.

Another popular method is to reserve a specific set of datatypes exclusively for Unicode storage, and to allow the length specification of columns using this datatype to be in multiples of Unicode characters[1]. Instead of declaring a database-wide Unicode character set, any columns created using this datatype would always contain Unicode data. This approach provides two distinct advantages over the former; firstly, it can always be guaranteed that Unicode data can be stored in any instance of a database, regardless of the character set specified upon database creation. Secondly, as column lengths are specified in terms of Unicode characters, there is no ambiguity as to how many Unicode characters can be stored in any particular column.

This approach is currently used by Microsoft SQL Server using NCHAR, NVARCHAR and NTEXT to indicate Unicode variants of the SQL CHAR, VARCHAR and TEXT datatypes.

Given that PeopleTools table columns are now declared by application developers in terms of Unicode characters, all SQL DDL statements generated by PeopleTools to reflect these in the database are not only specific to each database platform, but also differ depending on whether or not the current database is implementing Unicode. Should the database declare Unicode columns in multiples of bytes, the PeopleTools column size needs be multiplied by the worst-case ratio between Unicode characters and the number of bytes to represent a character in the transformation used by the database. Thus, with a database-level character set of UTF-8, a CHAR(10) column defined in PeopleTools requires a CHAR(30) column in the database to handle the worst-case ratio of 3 bytes per character. One significant danger in this case is that in a CHAR(30) database column could theoretically be updated by a user with access to the SQL interpreter to contain 30 Latin characters, even though the PeopleSoft definition expects only 10 Unicode characters. To workaround this situation, a database-level CHECK constraint is created to ensure that the character length of the column's contents is no longer than one third of the column's byte size.

The situation with Microsoft SQL Server is much simpler, as a NCHAR(10) column suffices to represent a 10 Unicode character PeopleSoft column definition.

Database Static Limits

When expanding database column sizes, it has to be expected that static object size limits imposed by the database management system will be approached. Specifically, once tripling character column sizes to accommodate UTF-8 data, PeopleTools hit the maximum row size limits on Sybase and Informix databases for several high-column-count tables. Similarly, the maximum index key length was also reached on Sybase for several indexes using long character keys.

PeopleSoft is continuing to work with these database vendors to increase these static limits, and in the meantime, can only support ANSI character storage in these database systems.

Database Application Programmatic Interface

Finally, the call-level API provided by each database management system needs to be able to accept Unicode bind variables, SQL constants and Unicode output without first converting via a local ANSI character set.

---

[1] More correctly, the datatype quantities are specified in multiple of UTF-16 *code units*. Thus, a Unicode surrogate character or composite character is treated as multiple "characters" when calculating the column length.

The simplest approach is to use an ODBC Version 3.0 driver. The ODBC 3.0 specification requires drivers to provide a wide-character API implementing UCS-2 as the wide character data type. This ensures that any ODBC 3.0 client can fetch data from the ODBC data source in Unicode UCS-2, regardless of the encoding used by the underlying database itself for storage or transport. Unfortunately however, many database vendors still do not provide ODBC 3.0 drivers, and provide only ODBC 2.0 compliant versions. While ODBC 2.0 drivers can successfully be used with an ODBC 3.0 driver manager and the ODBC 3.0 wide-character API, all conversation between the driver and the driver manager occurs in the native ANSI character set of the workstation, and therefore data loss will occur. Several databases originally slated for Unicode support in PeopleTools 8 could not be implemented as end-to-end Unicode solutions due to the lack of a version 3.0 ODBC driver.

Of course, an alternative to ODBC, many vendors provide proprietary call-level APIs. Varying from vendor to vendor, some of these APIs allow the client application (PeopleTools in this case) to choose the character set for the conversation, including UTF-8.

For PeopleTools 8, where possible, the PeopleTools database driver API uses ODBC or the database's native API to perform the entire database API conversation in UCS-2, regardless of the back-end database's character set. In this case, only the SQL DDL statements issued need to differ based on whether or not the database data itself is encoded in Unicode or in a local ANSI character set. As of writing, this was possible using ODBC 3.0 for SQL Server 7, and Oracle Net8i Client for Oracle 8i databases.

Alternatively, when UCS-2 presentation is not available via the vendor's database API, PeopleTools uses either UTF-8 or as a last resort, the local machine's ANSI character set. In this case, before any data is fetched from the API or passed to the API from PeopleTools, conversion between the API character set (UTF-8 or local ANSI) and UCS-2 has to occur. This conversion is limited to the database layer - all upper layers of PeopleTools operate exclusively in UCS-2.

Application Definition Migration

Finally, application definitions needed to be Unicode enabled, to provide application developers and users with a consistent approach to managing character data in SQL database definitions, user interface components and business logic authoring.

One significant advantage of the PeopleTools development environment is that it has always shielded the application developer from much of the complexities of the underlying computing environment on which the application is being developed. To accomplish this, PeopleTools maintains its own data dictionary containing definitions of not only application objects such as screens, menus, security and business logic, but also underlying database schema objects such as tables, columns and views. Although a PeopleTools-based application developer has access to a powerful scripting language, PeopleCode, for writing online and batch processing steps, this language was specifically limited to allow them to work at the character-level only, and does not provide low-level functionality such as bitwise operations or byte-based processing. If this functionality were previously available to a developer, it would be possible to write code using PeopleTools that could obfuscate the use of character datatypes in such a way that it would be very difficult to automatically change all character definitions to be multiples of Unicode characters.

It was a relatively easy step to change the underlying quantities used by the PeopleTools application definitions to equate to Unicode characters in place of native encoding bytes when processing application definitions referring to character strings. Specifically, changes were required to ensure that:

COPYRIGHT ©2000 PEOPLESOFT, INC.

- PeopleTools modules responsible for generating SQL DDL to reflect PeopleTools schema objects in the underlying database needed to be modified to produce SQL describing columns in terms of Unicode characters (or UTF-8 bytes, depending on what database system is being used).

- Any file I/O operations need to give the developer the choice of reading/writing Unicode files on a character-by-character basis, ANSI files on a character-by-character basis or ANSI files on a byte-by-byte basis.

- Interfaces to third-party products that are not yet Unicode enabled need to convert the internally held UCS-2 data to the local ANSI character set before communicating with the third-party products' API.

Automated SQL DDL Generation

An example PeopleSoft table definition is shown in the following diagram. This figure shows a fragment of a typical PeopleSoft "record" object, which is instantiated in the back-end database as a SQL table.

PERSONAL_DATA (Record)

Record Fields | Record Type

Num	Field Name	Type	Len	Format	H	Short Name	Long Name
1	EMPLID	Char	11	Upper		ID	EmplID
2	NAME	Char	50	Name		Name	Name
3	NAME_PREFIX	Char	4	Mixed		Prefix	Name Prefix
4	NAME_SUFFIX	Char	15	Upper		Suffix	Name Suffix
5	LAST_NAME_SRCH	Char	30	Upper		Last Name	Last Name
6	FIRST_NAME_SRCH	Char	30	Upper		First Name	First Name
7	ADDRESSH_SBR	SRec					
8	ADDR_OTR_SBR	SRec					
9	PHONE_SBR	SRec					
10	NATIONAL_ID	Char	15	Num		ID	National ID
11	PER_STATUS	Char	1	Upper		Per Status	Personnel Status
12	ORIG_HIRE_DT	Date	10			Hire Date	Original Hire Date
13	SEX	Char	1	Upper		Sex	Sex

Prior to PeopleTools 8, the field lengths shown for character columns represented character byte count maximums. Therefore, the NAME field could store up to 50 bytes of data; 50 single byte characters, 25 double-byte characters or any combination of the two. However this specification already differed based on the character set of the database system. If the database used a shifting DBCS character set, where each boundary within the string between single-byte and double-byte characters needed to be marked with a shift-in or shift-out byte, the maximum possible characters in this field would be considerably less than 25 in order to account for the additional shifting bytes.

Now that all character quantities are measured in terms of UCS-2 codepoints, the meaning of these column lengths is greatly different. When running against a Unicode database, the NAME field can now accommodate up to 50 non-composed, non-surrogate Unicode characters. As described previously, the SQL necessary to create the table to store this data varies per platform.

Prior to Unicode and PeopleTools 8, this table would be created in Oracle, SQL Server and Sybase databases using SQL similar to the following:

```
CREATE TABLE PS_PERSONAL_DATA
(EMPLID           CHAR(11) NOT NULL,
 NAME             CHAR(50) NOT NULL,
 NAME_PREFIX      CHAR(4) NOT NULL,
...
```

When creating a Unicode database with PeopleTools 8, this SQL will be markedly different. Specifically, it will look similar to the following statements, depending on database platform:

Microsoft SQL Server

```
CREATE TABLE PS_PERSONAL_DATA
(EMPLID           NCHAR(11) NOT NULL,
 NAME             NCHAR(50) NOT NULL,
 NAME_PREFIX      NCHAR(4) NOT NULL,
...
```

Oracle 8i

```
CREATE TABLE PS_PERSONAL_DATA
(EMPLID           VARCHAR2(33) NOT NULL
   CHECK (LENGTH(EMPLID) <= 11),
 NAME             VARCHAR2(150) NOT NULL
   CHECK (LENGTH(NAME) <= 50),
 NAME_PREFIX      VARCHAR2(12) NOT NULL
   CHECK (LENGTH(NAME_PREFIX) <= 4),
...
```

Microsoft SQL Server declares all Unicode datatypes as NCHAR, and measures column size in terms of UCS-2 codepoints. No change is required except to implement the new NCHAR datatype for character columns.

However, as can be seen from the Oracle example, specified column widths need to be tripled in order to accommodate a fixed number of Unicode characters per column, as Oracle measures column sizes in terms of bytes in the local character set. For UTF-8, the worst-case ratio between characters and bytes for non-surrogate, non-composed UCS-2 BMP characters is 3:1, resulting in a tripling of each column specification. However, as it would be possible to store 30 ASCII characters in 30 UTF-8 bytes, it is necessary to add a column constraint enforced by the database engine to limit the number of *characters* allowable in each column. The Oracle LENGTH( ) SQL function returns the number of *characters* in a string, and therefore, by adding CHECK constraints to each character column to verify that the character length of data in that column is no longer than one third its bytes, this effectively accomplishes the same column declaration as Microsoft's NCHAR Unicode datatype.

Unicode in PeopleCode

Similarly, the scripting language built into PeopleTools known as PeopleCode also changes the quantities by which character datatypes are measured. Any operations formerly operating on bytes now operate on Unicode characters. Take for example, this simple PeopleCode fragment:

```
Declare Function RemoveAccent PeopleCode NAME FieldDefault;

SetDefault(FIRST_NAME_SRCH);
SetDefault(LAST_NAME_SRCH);
&COMMA = Find(",", NAME);
&LAST_NAME = Substring(NAME, 1, &COMMA - 1);
&FIRST_NAME = Substring(NAME, &COMMA + 1, 50);
&COMMA = Find(",", FIRST_NAME_SRCH);
If &COMMA > 0 Then
    &FIRST_NAME = Substring(&FIRST_NAME, 1, &COMMA - 1);
Else
   &FIRST_NAME = Substring(&FIRST_NAME, 1, 50);
End-If;
RemoveAccent(&LAST_NAME, LAST_NAME_SRCH);
RemoveAccent(&FIRST_NAME, FIRST_NAME_SRCH);
```

In this program, an employee name entered in the PeopleSoft standard format ([lastname] [suffix],[prefix] [firstname] [middle name/initial]) is broken apart into components using the first comma found in the string as a delimiter. Prior to PeopleTools 8 and Unicode, the *Find( )* operation operated at a byte level, but observed double-byte character boundaries. It returned the byte offset within the column variable NAME at which the first comma character existed. This is subsequently passed to *substring( )* to fetch the first and last parts of the name into two separate temporary character variables. Each component is then passed into a user-written function, *RemoveAccent( )* that attempts to flatten European diacritics in a copy of the employee's name to aid in accent-insensitive searches.

Now that PeopleCode is completely Unicode based, all character constants are automatically interpreted as Unicode constants, and string operations work in terms of Unicode UCS-2 codepoints. Therefore, the *find( )* function will now search for the first occurrence of the Unicode character U+002C COMMA in the Unicode string NAME and return the offset measured in characters. The *substring( )* functions also accept Unicode characters as offsets where they previously accepted byte offsets, and all string arithmetic, such as performed in the construct COMMA - 1 is calculated using, and returns results in terms of, Unicode UCS-2 codepoints. By changing the base quantities used for string manipulation not only in the declaration of database storage space, but also in runtime code, a very high level of backward compatibility with the previous, non-Unicode PeopleTools releases has been achieved. In fact, it is very difficult if not impossible to write byte-oriented PeopleCode programs in PeopleTools 8; functions are simply not provided to the developer to access the individual bytes of character data.

One important exception to this rule is for file I/O operations. Although PeopleTools internally processes all data in Unicode, it cannot be expected that all file interchange operations with external systems can also take place in Unicode today. Therefore, when reading from and writing to files, developers do have access to the specific byte length of a string, and can control the exact number of bytes read from or written to a file. PeopleTools protects the developer from splitting a string not on a character boundary, and rounds the string down to the nearest complete character in case of a problem.

Memory Length vs. Database Length

One final difficulty encountered with Unicode in PeopleTools is the disparity between the length of characters in memory on the PeopleTools client or server systems, and the length of the same character string in the database management system. This problem is introduced because PeopleTools supports the use of non-Unicode encodings within the database management system, while data in memory is always stored in Unicode.

The issue arises when calculating string byte lengths. As described previously, PeopleTools 8 internally maintains all string maximum sizes in terms of Unicode characters. However, when declaring character storage in back-end database systems using a non-Unicode encoding, these column sizes are still measured in bytes. Therefore, a limited number of situations, it is possible that a character string may require more bytes to store in the database than in memory in PeopleTools. Specifically, this occurs frequently when dealing with EBCDIC-based double-byte character sets for database storage, which require shifting characters between each DBCS and SBCS boundary within a string. Take, for example, the following Japanese sentence:

私はPeopleSoftのﾄｰﾋﾞです。

In this case, the sentence contains a mixture of single-byte and double-byte characters when represented in most Japanese ANSI codepages. In Unicode UCS-2, the sentence takes 40 bytes to store, as it is comprised of 20 separate Unicode characters (the dakuten on the half-width katakana character "bi" counts as a separate character in this case). In UTF-8, it would also require 40 bytes, as both half-width and full-width Japanese characters require 3 bytes per character. In the Japanese PC-based Shift-JIS character set, this sentence would require 26 bytes to store, as it comprises of 6 full-width and 14 half-width characters. However, when converted to Japanese EBCDIC, and shift-in/out bytes are added each time the string changes from half-width to full-width characters, the ANSI byte length of the string grows to 32 bytes. The locations of the shift-in/out bytes are shown in the diagram below: The small square boxes indicate "shift-out" bytes indicating the start of a double-byte character string. The small circles indicate "shift-in" bytes indicating the start of a single-byte character string.

私はPeopleSoftのﾄｰﾋﾞです。

Given that the physical byte length of this string differs depending on whether the host database is encoded in UTF8, UCS2, Shift-JIS or Japanese EBCDIC, it is necessary to check the string length in the target encoding to ensure the length does not exceed the maximum length defined for the column in the database management system before allowing the input as a valid string. Failure to perform this check at runtime could result in SQL errors at save time.

SUMMARY

Although the implementation of Unicode across the PeopleSoft application product lines by way of PeopleTools was successful, there were many roadblocks that were initially unexpected. For a company that prides itself in developing not only platform-independent but also database management system independent applications, the lack of portability of existing Unicode implementations in compilers and database systems came as a surprise. Although these problems were worked around in this release by building compatibility layers in-between PeopleTools and each target system, it is hoped that as the Unicode standard matures and becomes more widely adopted, vendors with cooperate in designing standard ways of implementing Unicode across platform, technology and corporation boundaries.

For PeopleSoft's international customer base, the move to Unicode in the PeopleSoft product line could not have come at a more opportune time. In today's world where global companies are adopting consolidated, worldwide eBusiness systems, the value of maintaining central mission critical systems while still fully supporting the business and linguistic needs of each local operation cannot be underestimated. Particularly when looking at applications that can be accessed using purely a web browser as a client, the idea of a single company-wide data repository can become a reality. With PeopleTools 8 and Unicode, our customer base can take advantage of a wider choice of architectures when planning their global application rollout, while preserving the ability to store and manage data in the local language of each of its global operations.

Unicode Portability Limitations

One of the main problems identified by this large-scale Unicode implementation is the lack of cross-platform portability of existing implementations of the Unicode standard in common software products, specifically C / C++ compilers and SQL-based relational database management system.

- SQL Databases
  Although the ANSI SQL standard defines how SQL databases define and manage character data, there is still no common implementation architecture for Unicode in database management systems. As can be seen from the selection of databases described in this paper, a wide range of techniques are used in database implementations today to declare and manage Unicode SQL datatypes. While some vendors have chosen to use a new datatype for Unicode data, and to size columns in terms of UCS-2 codepoints, others have treated Unicode as just another codepage, and continue to provide byte-based support for Unicode datatypes.

  An additional problem is the use of Unicode identifiers and constants in SQL statements themselves. A range of solutions currently exist, including using special identifiers to indicate quoted strings that should be represented as Unicode data, to presenting the entire SQL statement, including constants as a Unicode string.

- The ODBC 3.0 API goes a long way along the path of standardizing end-to-end Unicode access to SQL databases, but a common ODBC specification still does not solve the problem of schema definition. Work is needed between the major SQL database vendors to standardize not only the declaration of Unicode datatypes in DDL, but also mechanisms for quoting Unicode constant strings in SQL statements, and for fetching Unicode data via the call-level API. Additionally, vendors need to realize the importance of end-to-end Unicode communication. Unlike older multiple-character set database implementations, where transcoding via local ANSI codepages could maintain integrity of character data, it is now critical that database applications can fetch Unicode data directly from the back-end Unicode database without having to convert via a non-Unicode character set in the process.

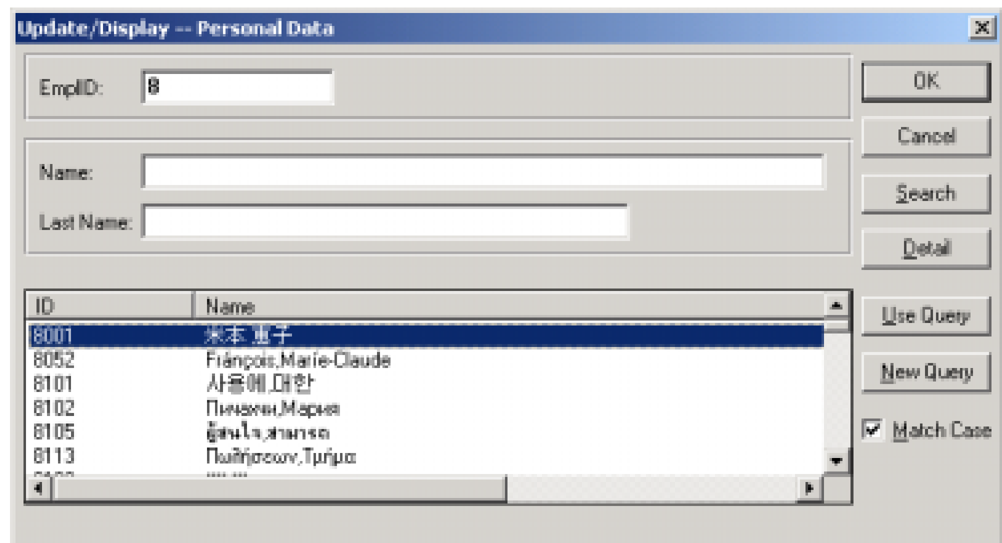
- C / C++ Language
  Although the ANSI C language provides a good mechanism for handling wide character data using the wchar_t datatype and its related wide character string functions, each vendor is free to implement their own choice of wide character representation. Of the 11 operating systems and compilers examined by PeopleSoft during the development and porting work for PeopleTools 8, no fewer than 4 different approaches to encode wchar_t wide character types were encountered. Given the continued byte orientation of the C language, it is important for any major cross-platform development work to be able to rely on a standard implementation of wide characters, and more specifically, Unicode, across each operating system and compiler version. 3rd party Unicode library vendors such as Basis Technology Corp. go a long way in solving this problem with Unicode libraries such as Rosette, however cooperation between vendors is needed to ensure a consistent and reliable standard of Unicode representation across each compiler system. One example of a difficulty imposed by the non-standardization of wide character datatypes that cannot easily be solved by implementing a 3rd-party Unicode library such as Rosette is the presentation of wide character constants in C / C++. All ANSI-compliant C and C++ compilers provide the ability to quote constant character strings to mark them as wide character constants using the L"string" method. However, when the wide character type being used by the application does not match with the default wide character type of the compiler, it is incredibly difficult to widen these string constants at compile-time. PeopleSoft invested a great deal of time and effort in building a pre-compiler to widen such quoted strings, however as it not a part of the compiler itself, this additional step is very intrusive and problematic.

Internal PeopleSoft Benefits of Unicode in PeopleTools 8

The obvious benefit of implementing Unicode for PeopleSoft internally is the myriad of languages that can be enabled for support in PeopleTools 8 with a single set of engineering. Instead of adding support for more individual legacy character sets, where each new character set supported may provide access to only one or a few languages, the Unicode implementation opens the door for PeopleTools-based applications to be used virtually worldwide. Supporting all these languages with a single set of binaries also greatly lowers the cost of supporting international versions of the PeopleTools product suite. Now, the only difference between PeopleTools versions released in each international market are the language of the user interface translations, and even these translations are all included on a single CD-ROM, and can be installed and re-configured at will.

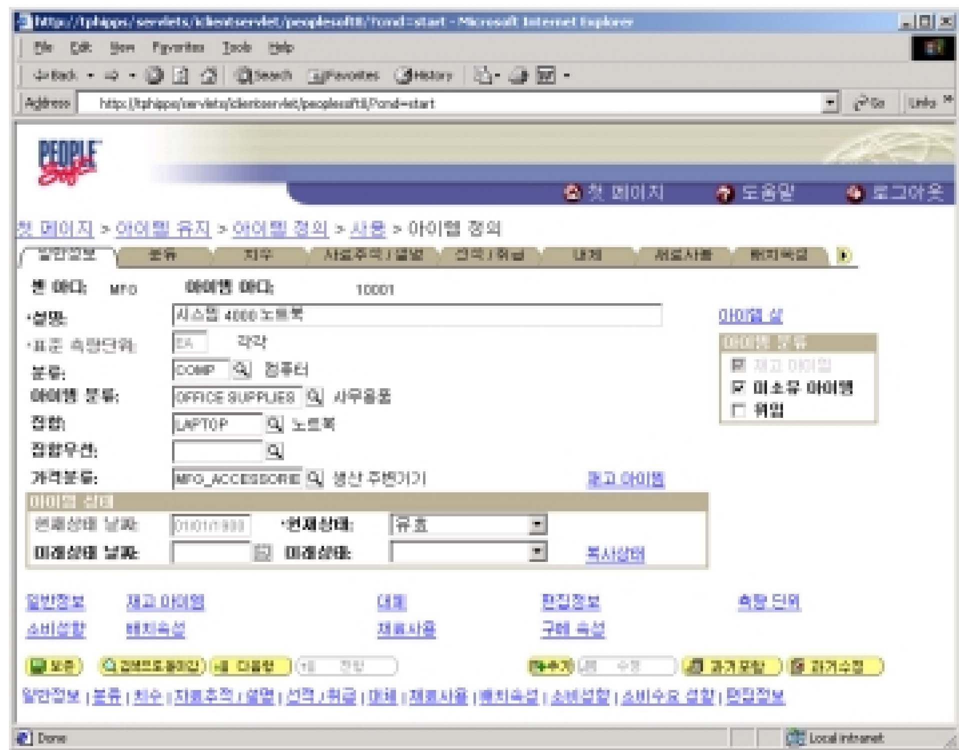
End-User Benefits of Unicode in PeopleTools 8

More importantly, the implementation of Unicode in PeopleTools 8 allows our largest customers to realize the ability to maintain a truly worldwide view of their organization in a single PeopleSoft system. As described previously, many PeopleSoft customers were faced with the reality of installing and managing dozens of separate PeopleSoft databases in order to be able to support the local languages of each of their worldwide operations.



As can be seen from the Windows screen-shot above and the Web screen shot below, it is now very easy with PeopleTools 8 to maintain a single database encoded in Unicode, and view and maintain data in all languages simultaneously. This centralized database model is becoming more prevalent with the introduction of very lightweight client versions of PeopleSoft, where users can access a system with only a web browser over a dial-up connection.

The ability to build an application and deploy it against a database system that uses a legacy encoding or a Unicode transformation is one more level of portability offered by the PeopleTools suite. Although the database system may rely on a national or language specific encoding, PeopleTools continues to perform all processing in Unicode, eliminating the need of managing many disparate encodings of data in a large, production system.



COPYRIGHT ©2000 PEOPLESOFT, INC.