Oracle Tuxedo Globalization Features: Multibyte Support for the Asia Pacific Region

An Oracle White Paper
Updated June 2008

Oracle Tuxedo Globalization Features: Multibyte Support for the Asia Pacific Region

Oracle Tuxedo Globalization Features: Multibyte Support for the Asia Pacific Region

INTRODUCTION

Internationalization and localization are integral components of the Oracle Tuxedo system. For the Asia Pacific region, the software provides full support for multibyte character codeset handling and enables developers to build a solution with no language limitations.

Internationalization and localization are integral components of the Oracle Tuxedo system. For the Asia Pacific region, the software provides full support for multibyte character codeset handling and enables developers to build a solution with no language limitations. By eliminating the need to customize software to achieve internationalization, Oracle Tuxedo allows companies to easily extend their applications to employees and partners in multiple languages.

This paper describes the globalization features of Oracle Tuxedo, illustrating functionality through examples.

GLOBALIZATION ENHANCEMENTS IN ORACLE TUXEDO

Oracle Tuxedo provides the following internationalization enhancements:

- Support for multibyte character typed buffers for user data
- The capability for programmatic conversion on demand using the APIs or automatic conversion between Chinese, Japanese, and Korean codeset encodings
- The ability to "get" and "set" codeset encoding information and to turn automatic conversion on and off—both programmatically and administratively
- Support for easy replacement of conversion library with custom conversion functionality

These specific enhancements use several system features, including a typed buffer called MBSTRING, a field type called FLD_MBSTRING, and a multibyte character transport and convert API.

With Oracle Tuxedo, programmers can manage encoding conversions both administratively, by using the environment variables TPMBENC and TPMBACONV, and programmatically, by using new API functions. The ability to turn automatic conversion on and off programmatically allows an application to limit conversions to only when they are required—providing good control over conversion-related performance.

If the Oracle Tuxedo system is configured for automatic codeset encoding conversion, when an MBSTRING buffer (or an FLD_MBSTRING field in an FML32 buffer) is transmitted between processes running on different computer platforms, the underlying system converts from one codeset encoding to another. Specifically, the receiving side automatically converts the MBSTRING buffer from the sender's codeset encoding representation to the receiver's codeset encoding. If automatic codeset conversion is not configured manually through the environment variables TPMBENC and TPMBACONV, the sending or receiving application can request codeset encoding conversion on a case-by-case basis using the conversion APIs (see Appendix 1 for specifics).

Use of the GNU iconv conversion library provides common codeset-conversion functionality across the UNIX and Windows platforms. Use of Oracle Tuxedo typed buffers allows easy replacement of the conversion library with custom functionality, for example, for testing or performance tuning.

A PERSPECTIVE ON SOFTWARE GLOBALIZATION

Most software products, such as operating systems, libraries, and development tools, are designed and developed for international environments—environments that have very different linguistic, cultural, and presentation requirements. For example, a large corporation with headquarters in Tokyo and branches in New York and Seoul might require a combination of English, Japanese, and Korean software environments. In addition, these internationally distributed computing environments must also support location-based changes in time, numeric values, dates, monetary formats, message representation, and codeset encoding schemes. They must support all of these requirements spontaneously (without restarting the application) as transactions span global locations. Software that meets these requirements is called globalized software.

Internationalization and Localization

You achieve software globalization by addressing requirements for both internationalization and localization. Internationalization makes software portable between regions in which different spoken languages and customs are used. To create internationalized software, the developer isolates the parts of a program that depend on language and culture. For example, error messages are isolated for easy translation to the language of the locale in which they are read. A locale is a geographic or political region that shares the same spoken language and customs. The internationalized program is either designed or adapted to pick up the locale-dependent pieces during system initialization.

Internationalization makes software portable between regions where different languages and customs are used. Localization allows for specific versions within the internationalized program to be used within a geographic or political region.

Localization is the process of creating locale-specific versions, or packages, of the locale-dependent pieces. Localization includes the translation of text such as labels in the user interface, error messages, and online help. It also includes the culture-specific formatting of data items such as time, monetary values, dates, and numbers. Oracle develops packages for localization that are made available to customers who need them.

Codesets and Encoding

A *character set* is a set of elements that represent text in a given spoken language. The English alphabet is a character set. There might be an implied ordering relationship between the characters, but the characters are not assigned specific values. For example, you might recite the English alphabet starting with *a, b, c,* and continue in a customary way until you finish with *x, y, z*. Although there is this implied ordering, there is no numeric relationship between the characters that implies this. A codeset provides such a numeric relationship, giving computer programs a mechanism for manipulating the character set.

A *codeset,* also called a coded character set, is a computer-based mapping of characters, to unique non-negative integers. The mapping of unique binary values for a codeset is called an encoding for that codeset. In the United States, ASCII and Unicode are two codesets that represent the set of characters on most computer keyboards. ASCII is also an encoding. There can be multiple encodings for a particular codeset. For example, in Japan, computer vendors support at least three encodings for Kanji, a Japanese codeset: EUC-JP, Shift-JIS (SJIS), and ISO-2022-JP. Most UNIX vendors support EUC-JP and some also support SJIS. Windows, OS/2, and Macintosh support SJIS. In Korea, the KSC5601 encoding is widely used, whereas in China, GBK is used. Java supports its native Unicode and its many foreign encodings.

Multibyte Encoding Conversion: Past and Present

Oracle Tuxedo supports multibyte codeset handling functionality. Although standard English can be accommodated with an 8-bit (single byte) codeset encoding scheme, Chinese, Japanese, and Korean languages require a multibyte codeset-encoding scheme.

The alphabetic characters in European languages, including standard English, can be accommodated with an 8-bit (single byte) codeset encoding scheme. However, Chinese, Japanese, and Korean languages, which are based on a large set of symbols (or ideographs), require a multibyte codeset-encoding scheme. Oracle Tuxedo supports these Asia Pacific region character sets with multibyte codeset handling.

Before Oracle Tuxedo, the application developer had to create a custom conversion solution to get globalization features. However, custom conversions can handle only very specific use cases. For example, one custom solution might handle the conversion of SJIS to EUC-JP, while another converts between SJIS and ISO-2022-JP. There is no need to develop these types of custom conversions with Oracle Tuxedo.

It is important to remember that Oracle Tuxedo's codeset conversion capability is designed for conversions between encodings for a codeset (for example, between the encodings UTF-8 and UTF-16BE for the Unicode codeset). It is not a conversion between codesets (for example, between ASCII and Unicode) or a translation between languages, but a conversion between different encodings for the same language.

Typical Conversion Scenarios

One very common multibyte conversion scenario involves different Kanji encoding schemes running on different platforms (for example, client/server systems). When clients and servers are hosted on platforms that use different encoding schemes, it is necessary to perform encoding conversions between the platforms. The example shown in Figure 1 illustrates a scenario for a distributed computing environment in Japan. In the example, a client resides on a Windows machine that supports SJIS. The Oracle Tuxedo server machine is UNIX based and supports EUC-JP.

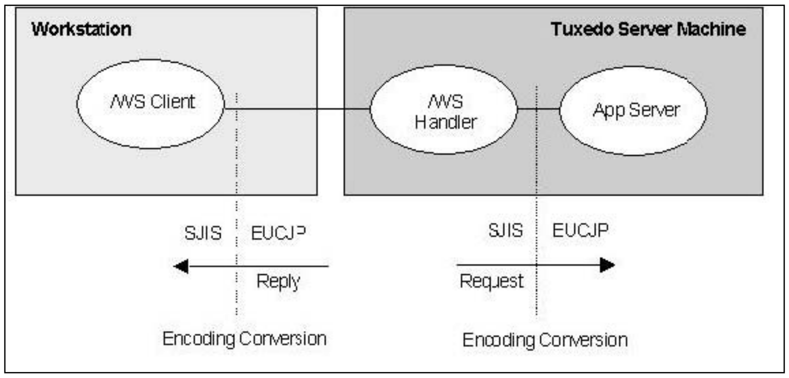

Figure 1: A typical scenario of a distributed computing environment

ORACLE TUXEDO MULTIBYTE CONVERSION FUNCTIONALITY

In the previous example, multibyte character application data is transported between the Windows client and Oracle Tuxedo server processes using a typed buffer called MBSTRING. A set of API functions associated with this typed buffer determine the codeset encoding name and perform multibyte data conversion using the GNU iconv library.

Oracle Tuxedo developers implemented the MBSTRING buffer by adding a new entry to the typed buffer switch structure (tm_typesw). This allows Oracle Tuxedo to determine which routines to call for each typed buffer. For the MBSTRING buffer, the system calls the internal function _mbsconv() to perform automatic codeset multibyte data conversion. This internal function then uses the GNU library routines to convert the user data.

Conversion is inherently costly in terms of performance. To prevent encoding conversions from negatively impacting performance, Oracle Tuxedo allows the user to control conversions both administratively and programmatically.

Managing Encoding Conversion

There are two ways of controlling encoding conversions: administratively, by using the environment variables TPMBENC and TPMBACONV, and programmatically, by using the API functions. If the environment variables are set for automatic conversion, the receiving Oracle Tuxedo system converts the data in the buffer from one encoding to another. Otherwise you can use the programmatic interface tuxsetmbaconv() to turn automatic encoding on and off without restarting the application, and thereby limit conversions to occur only when they are required. Otherwise conversions can take place at each hop, which deteriorates performance.

Figure 2 illustrates the same example described in Figure 1, but it is expanded to show details about how Oracle Tuxedo handles multibyte data. The environment variables TPMBENC and TPMBACONV are set on each machine to identify the encoding and the state (on or off) of automatic encoding conversion. This example, set in a Japan locale, illustrates a Windows client that supports SJIS encoding and a UNIX server that supports EUC-JP encoding. The typed buffer header identifies the buffer as an MBSTRING type and provides encoding and data length information. The buffer itself holds user data represented in the encoding identified in the header. The client request buffer holds data represented by SJIS encoding, and the server reply buffer holds data represented by EUC-JP encoding.

There are two things to consider when designing an application. First, conversion is inherently costly in terms of performance. Use of automatic conversion, in the example here, will mean that conversion is done twice for a message—once when a request is received by the server and then again when the reply is received by the client. Second, the size of the user data in the buffer will change depending upon conversion. On the client side in this example, the buffer will either be the same size after the conversion or it will be smaller. On the server side, the buffer size will be the same or grow.

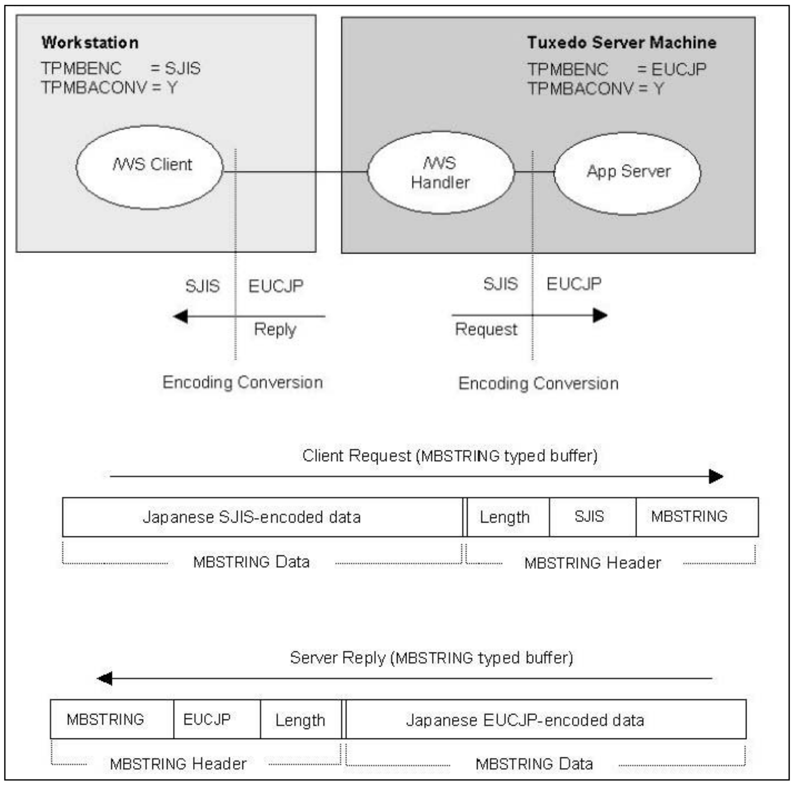

Figure 2: Data conversion using an MBSTRING typed buffer

Processing on the Client-Side

When the client process is invoked, it retrieves or sets the name of the codeset encoding supported by the machine it is running on. For example, to retrieve the encoding name set using the environment variable TPMBENC, the client calls tuxgetmbenc() to search the environment list for a string of the form TPMBENC=*encodingName*. If the string is present, the encoding name is passed along with the user data when the client calls Oracle Tuxedo's tpalloc() to allocate a new MBSTRING buffer. The encoding name is then cached, so the call need only be done once during the initial process invocation of the typed buffer switch function. If the environment variable TPMBENC is not defined, or if you want to reset it during processing, the application can use API functions to accomplish this.

Once the client calls tpalloc, Oracle Tuxedo provides buffer allocation and data conversion, as illustrated in Figure 3. The underlying Oracle Tuxedo system allocates memory for the new MBSTRING buffer and uses an internal version of the tuxgetmbenc() function to get the encoding name defined for the TPMBENC environment variable, if it is set. Oracle Tuxedo adds the encoding name to the MBSTRING buffer header and returns the allocated buffer to the client.

Later, when the client sends the MBSTRING buffer—for example, using tpsend() or tpcall()—Oracle Tuxedo will again intervene to perform conversion on the receiving side, as described in the next section, "Processing on the Server-Side."

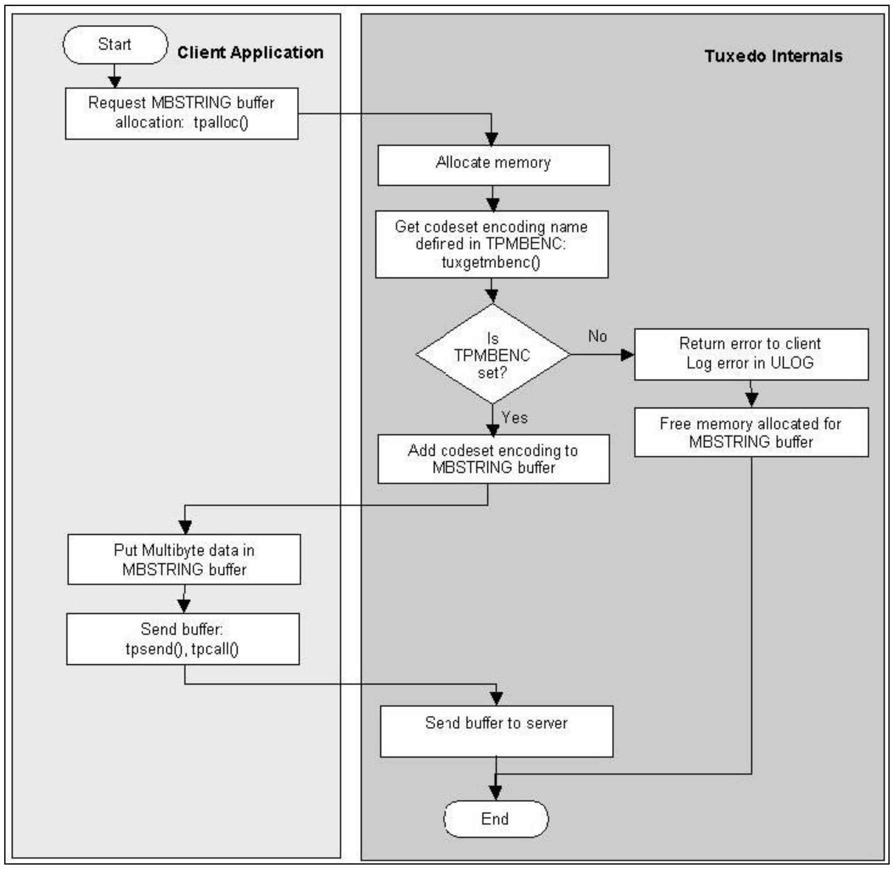
Start
Client Application
Tuxedo Internals
Request MBSTRING buffer allocation: tpalloc()
Allocate memory
Get codeset encoding name defined in TPMBENC: tuxgetmbenc()
Is TPMBENC set?
No
Return error to client Log error in ULOG
Free memory allocated for MBSTRING buffer
Yes
Add codeset encoding to MBSTRING buffer
Put Multibyte data in MBSTRING buffer
Send buffer: tpsend(), tpcall()
Send buffer to server
End

Figure 3: Client processing with Oracle Tuxedo providing buffer allocation and data conversion

Processing on the Server-Side

The flow diagram in Figure 4 shows the underlying Oracle Tuxedo processing that takes place when the client sends a request to a server that includes an MBSTRING typed buffer. Note that Figure 4 also lists the same steps used when the client receives a reply. Before passing the message on to the service, Oracle Tuxedo receives the MBSTRING buffer. It checks the environment variable TPMBACONV to determine if automatic conversion is set. If it is not, Oracle Tuxedo delivers the data in the MBSTRING buffer to the server without encoding conversion. If automatic conversion is set, Oracle Tuxedo retrieves the encoding name defined in TPMBENC. It does some error checking to ensure that the encoding value is set because otherwise it cannot do the conversion. If the value were not set, it would log an error and pass control to the server.

If the TPMBENC environment variable is set, Oracle Tuxedo's type switch element automatically compares the client's encoding name with the server's and, if the encoding names are different, Oracle Tuxedo automatically converts the encoding of the incoming message to the encoding supported by the server's machine using GNU iconv-based library routines or on a user-created custom conversion routine. (See the "Customization" section for more information about creating custom routines.) Oracle Tuxedo delivers the converted data to the service and passes control to it.

For more examples of server-side and client-side multibyte conversion applications, please see Appendix 2 at the end of this document.

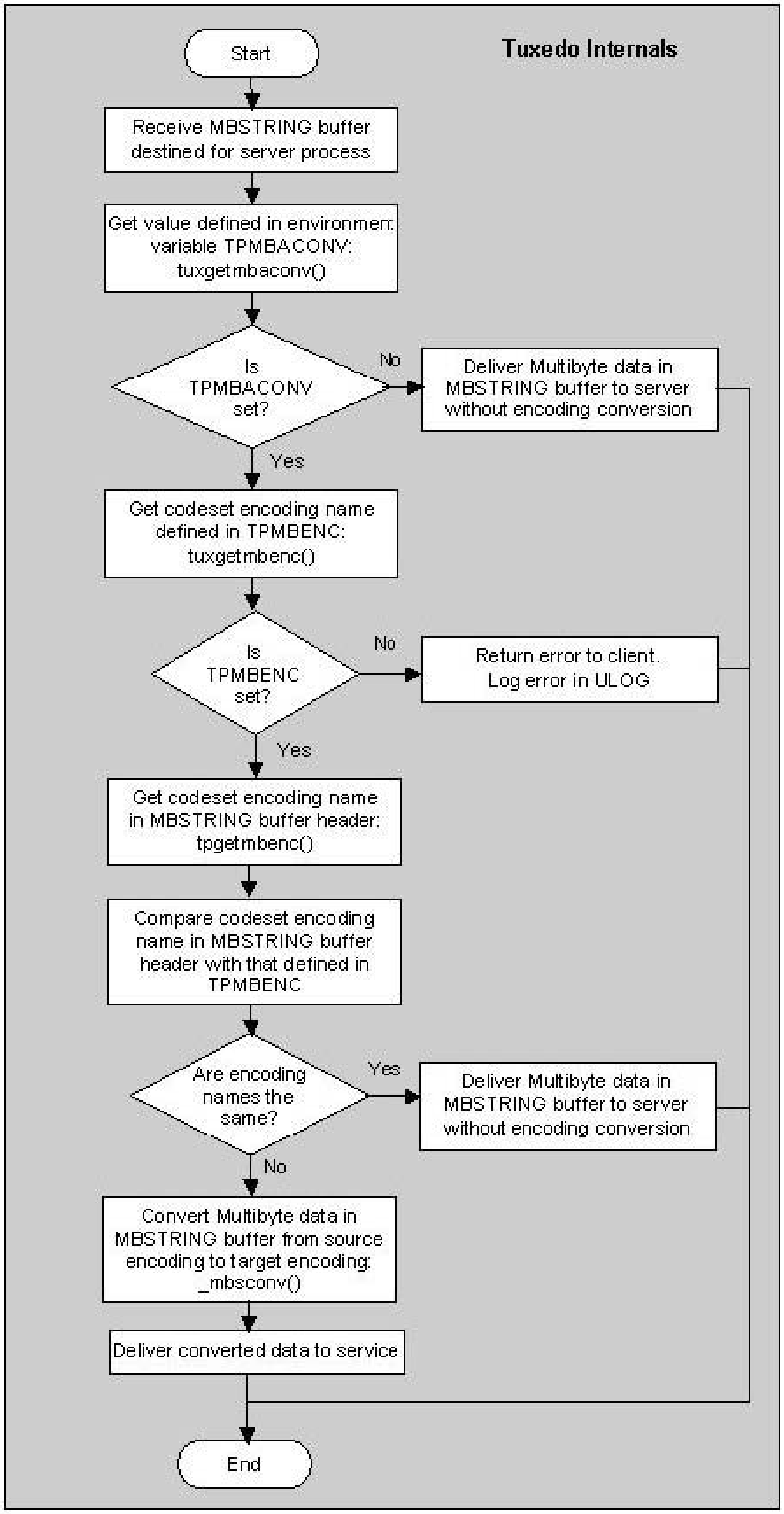
Tuxedo Internals
Start
Receive MBSTRING buffer destined for server process
Get value defined in environment variable TPMBACONV: tuxgetmbaconv()
Is TPMBACONV set?
No
Deliver Multibyte data in MBSTRING buffer to server without encoding conversion
Yes
Get codeset encoding name defined in TPMBENC: tuxgetmbenc()
Is TPMBENC set?
No
Return error to client. Log error in ULOG
Yes
Get codeset encoding name in MBSTRING buffer header: tpgetmbenc()
Compare codeset encoding name in MBSTRING buffer header with that defined in TPMBENC
Are encoding names the same?
Yes
Deliver Multibyte data in MBSTRING buffer to server without encoding conversion
No
Convert Multibyte data in MBSTRING buffer from source encoding to target encoding: _mbsconv()
Deliver converted data to service
End

Figure 4: Server processing showing the underlying Oracle Tuxedo processing that occurs when the client sends a request to a server that includes an MBSTRING typed buffer

You can easily install a custom automatic conversion routine for MBSTRING by replacing the name of the default conversion function with the name of the custom function in the definition for MMSTRING.

Customization

A developer might want to create a custom conversion function if, for example, better performance is needed to instrument or debug the use of the conversion functions, or if there is a requirement for custom characters that the provided libraries do not handle. You can easily install a custom automatic conversion routine for MBSTRING by replacing the name of the default conversion function with the name of the custom function in the definition for MBSTRING. MBSTRING is defined in the tmtypesw.c file, which is where the Oracle Tuxedo typed buffers are added to the process buffer type switch (tm_typesw). You can convert a buffer independent of the automatic conversion capability using a tpconvmb() function at the application level.

The following fragment from the tmtypesw.c file describes a custom definition for MBSTRING. The last line shows that the name of the default conversion function, _mbsconv, has been replaced with the name of the custom function, CUSTmbconv. Thus the custom conversion routine will be called instead of the default function when Oracle Tuxedo performs automatic encoding conversion for MBSTRING type data.

```
"MBSTRING",     /* type */
"*",            /* subtype */
0,              /* dfltsize */
_mbsinit,       /* initbuf */
NULL,           /* reinitbuf */
NULL,           /* uninitbuf */
NULL,           /* presend */
NULL,           /* postsend */
NULL,           /* postrecv */
NULL,           /* encdec */
NULL,           /* route */
NULL,           /* filter */
NULL,           /* format */
NULL,           /* presend2 */
CUSTmbconv      /* customized multi-byte codeset conversion */
```

The CUSTmbconv code consists of the functions normally used for conversion, but this is reduced to the iconv calls that are normally used on a UNIX operating system. The sample customization function is available in Appendix 2.

Encoding Alias Names

The GNU iconv specification permits usage of a charset.alias file. This file allows a user to define an alias for an existing encoding name. Such capability is in addition to built-in lists that GNU has for some common names used to specify a unique encoding, such as SJIS, SHIFT_JIS, SHIFT-JIS, MS_KANJI, CSSHIFTJIS, and so on. Although this capability is available, there is a performance cost, so it is not recommended. Instead choose one of the names from the GNU iconv specification to use for your encoding.

MULTIBYTE DATA WITH FIELD MANIPULATION LANGUAGE BUFFERS

With Oracle Tuxedo, field manipulation language 32 (FML32) buffers accept a FLD_MBSTRING field type for codeset-identified multibyte data. Fmbpack32() and Fmbunpack32() functions provide this field with the information needed for processing it. The packed data is sent with an FML32 buffer and the receiver of the FML32 buffer; if the TPMBACONV environment variable is set, it automatically executes the FML32 buffer type switch conversion function (_fmbconv32). This function checks the FML32 buffer for FLD_MBSTRING fields and performs conversion if the encoding name within the field information is not the same as the local TPMBENC environment variable. As with the _mbconv function, a user can customize by redefining the tmtypesw. An application accesses converted packed data from the FML32 buffer using FML32 API functions and the FLD_MBSTRING field type. It unpacks data with the Fmbunpack32() function.

See Appendix 2 for examples of using MBSTRING and FLD_MBSTRING with FML32 buffers.

CONCLUSION

Globalized software applications that are portable between regions with different spoken languages and customs are critical to the successful operation of a global company.

Globalized software applications that are portable between regions with different spoken languages and customs are critical to the successful operation of a global company. Before Oracle Tuxedo, the application developer had to create a custom conversion solution to get globalization features. However, custom conversions can handle only very specific use cases. The globalization capabilities provided by Oracle Tuxedo include

- Support for multibyte character typed buffers for user data
- The capability for programmatic conversion on demand using the APIs or automatic conversion between Chinese, Japanese, and Korean codeset encodings
- The ability to “get” and “set” codeset encoding information and to turn automatic conversion on and off—both programmatically and administratively
- Support for easy replacement of conversion library with custom conversion functionality

These globalization features enable application administrators to more easily maintain applications in multiple languages.

APPENDIX 1: APIS ASSOCIATED WITH MULTIBYTE DATA

The tables in this appendix show functions associated with the MBSTRING and the FLD_MBSTRING commands.

tpconvmb()	**Converts characters from an encoding passed along with an input buffer to a named target encoding.**
tpgetmbenc()	**Allows a client or server process to retrieve or reset the codeset encoding name from an MBSTRING buffer. tpsetmbenc() returns a value indicating whether the encoding name is set or not. Use tpsetmbenc() if the encoding name needed by the application is different than the one specified as part of the MBSTRING buffer.**
tuxsetmbaconv()	**Allows a client or server process to get or set the TPMBACONV environment variable. If the get operation returns a value indicating the TPMBACONV is set, then codeset data conversions will be executed automatically by the buffer type switch functions. Executing the tuxsetmbaconv() function will set or unset the TPMBACONV function.**
tuxsetmbenc()	**Allows a client or server process to get or set the TPMBENC environment variable. The application can use the set function to set or reset TPMBENC. The get function searches the environment list for a string of the form TPMBENC=*value*. If present, it returns a pointer to the value in the current environment.**

Table 1: MBSTRING associated functions

Fmbpack32()	**Creates a byte stream for use as input to FML32 API functions. It takes as inputs the codeset encoding name, the codeset multibyte data, and the length of the input data. It returns an output data pointer containing the above inputs in a format that FML32 can use.**
Fmbunpack32()	**Takes the output of FML32 API functions actions on FLD_MBSTRINGs and converts it into information that an application can use. It takes as input the packed byte stream resulting from the FML32 function and the number of bytes. It returns the codeset encoding name, the multibyte user data, and the returned data length.**
tpconvfmb32()	**Permits the application developer to execute a multibyte data conversion separate from the typed buffer switch function. It takes an input FML32 buffer, an output FML32 buffer, and a target codeset encoding name. It walks through the input FML32 buffer and update FLD_MBSTRING field types that contain a codeset encoding name different from the target encoding name argument.**

Table 2: FLD_MBSTRING associated functions

APPENDIX 2: SOFTWARE EXAMPLES

A Multibyte Data Conversion Example

This example illustrates the use of API functions associated with MBSTRING in a simple conversion scenario. A multibyte data conversion example, described in the "Customization" section, provided the application perspective for this example.

Client-Side Application

```
/* #ident "@(#)apps:simpapp/simpclmb.c 1.1" */

#include <stdio.h>
#include "Uunix.h"
#include "atmi.h" /* TUXEDO  Header File */
#if defined(__STDC__) || defined(__cplusplus)
main(int argc, char *argv[])
#else
main(argc, argv)
int argc;
char *argv[];
#endif
{
/*

     *********************************************************
     This example will send an input string to a service
     TOUPPERMB that will convert the characters to uppercase
     and then return the result back to this client. This
     client-side process encoding name will be defined to be
     UTF-16LE, the buffer to be sent will be redefined to the
     UTF-8 encoding and the server-side encoding will be UTF-
     16BE. If automatic conversion is turned on for both sides,
     then the server process will convert the MBSTRING from
     UTF-8 to UTF-16BE before passing it on to the TOUPPERMB
     service. After the service is done and returns the
     MBSTRING, it will be converted at this client process
     from UTF-16BE to UTF-16LE(because that is the defined
     encoding for this process) before delivering the resulting
     buffer as the tpcall rcvbuf argument to this application.
     Finally the rcvbuf will again be converted to the UTF-8
     encoding and printed out. The UTF-16LE steps are not
     needed but are added to show some API usage. (ie UTF-
     8<=>UTF-16BE could have been by auto conversion)
     *********************************************************
*/
     char *sendbuf, *rcvbuf;
     long sendlen, alloclen, rcvlen;
     int ret,iolen;
     if(argc != 2) {
         (void) fprintf(stderr, "Usage: simpclmb string\n");
         exit(1);
     }
     /* Attach to System/T as a Client Process */
     if (tpinit((TPINIT *) NULL) == -1) {
         (void) fprintf(stderr, "Tpinit failed\n");
         exit(1);
     }


/*
```

```
        ********************************************************
        If it is desired to have automatic multibyte conversion
        "OFF" then comment out, or delete, the following six
        lines.
        The tuxsetmbaconv will only control this client process.
        The server process will need to set its own environment
        variable or execute its own tuxsetmbaconv() function.
        NOTE:An alternative to using these two lines is to set
        the TPMBACONV
        environment variable(eg export TPMBACONV="YES").
        ********************************************************
   */
        ret = tuxsetmbaconv(MBAUTOCONVERSION_ON,0);
        if(ret == -1) {
            (void) fprintf(stderr, "tuxsetmbaconv failed\n");
            exit(1);
        }
        (void) fprintf(stderr, "tuxsetmbaconv ON done.\n");
        /*
        ******************************************************
        NOTE:An alternative to using the following six lines is
        to set the TPMBENC environment variable(eg export
        TPMBENC="UTF-16LE").
        ******************************************************
        */
        ret = tuxsetmbenc("UTF-16LE",0);
        if(ret == -1) {
            (void) fprintf(stderr, "tuxsetmbenc failed\n");
            exit(1);
        }
        (void) fprintf(stderr, "tuxsetmbenc UTF-16LE done.\n");
        sendlen = strlen(argv[1]);
        /*

        ********************************************************
          NOTE: This example is using an ASCII input string. The
          customer specific encoding used may not work well with
          the OS string functions due to an embedded NULL in the
          character definition. In general the memory or wcstring
          functions can be used without being concerned about the
          codeset encoding having embedded NULLs. Therefore 1 is
          not added to sendlen, in this example, for a NULL
          terminator. Only exact bytecnt is used. It is left to
          the developer to use the string functions and add the
          NULL terminator to the send length.
        ********************************************************
        */
        (void) fprintf(stderr,"Input: %s, Length: %d\n", argv[1],
        sendlen);
        /* Allocate MBSTRING buffers for the request and the
        reply */ alloclen = sendlen * 4; /*max size buf ensures
        min # iconv iterations*/ if((sendbuf = (char *)
        tpalloc("MBSTRING", NULL, alloclen)) == NULL) {
       (void) fprintf(stderr,"Error allocating send buffer: %s\n",
        tpstrerror(tperrno));
             tpterm();
             exit(1);
        }
        if((rcvbuf = (char *) tpalloc("MBSTRING", NULL,alloclen))
        == NULL) {
             (void) fprintf(stderr,"Error allocating receive
          buffer\n");
        tpfree(sendbuf);
        tpterm();
          exit(1);
```

```
      }
/*
     *********************************************************
     The default encoding for the newly tpalloc'd send buf is
     UTF-16LE (because we did a tuxsetmbenc() above) but the
     data that is being input to this client is UTF-8
     encoding(ie argv[1] in UTF-8). So I need to reset the
     sendbuf encoding to UTF-8.
     *********************************************************
*/
     ret = tpsetmbenc(sendbuf,"UTF-8",0);
     if(ret == -1) {
     (void) fprintf(stderr, "tpsetmbenc UTF-8 failed\n");
     (void) fprintf(stderr, "Tperrno = %d\n", tperrno);
     exit(1);
     }
     (void) fprintf(stderr, "tpsetmbenc UTF-8 done.\n");
     (void) memcpy(sendbuf, argv[1], (size_t)sendlen);
     /* Request the service TOUPPERMB, waiting for a reply */
     ret = tpcall("TOUPPERMB", (char *)sendbuf, sendlen, (char
     **)&rcvbuf, &rcvlen, (long)0);


     if(ret == -1) {
          (void) fprintf(stderr, "Can't send request to
          service TOUPPERMB\n");
          (void) fprintf(stderr, "Tperrno = %d\n", tperrno);
          tpfree(sendbuf);
          tpfree(rcvbuf);
          tpterm();
          exit(1);
     }
     (void) fprintf(stdout, "Returned rcvbuf Length %d\n",
     rcvlen);


/*
     *********************************************************
     The rcvbuf was automatically converted from UTF-16BE to
     UTF-16LE, (because we used tuxsetmbaconv() initially) when
     this process received the reply buffer from the TOUPPERMB
     service, but this application requires it to be printed
     out using UTF-8 encoding so force another conversion from
     UTF-16LE to UTF-8.
     *********************************************************
*/
     iolen = (int)rcvlen;
     ret = tpconvmb(&rcvbuf, &iolen, "UTF-8", (long)0);
     if(ret == -1) {
          (void) fprintf(stderr, "Can't execute tpconvmb.\n");
          (void) fprintf(stderr, "Tperrno = %d\n", tperrno);
          tpfree(sendbuf);
          tpfree(rcvbuf);
          tpterm();
          exit(1);
     }


/*
     *********************************************************
     NOTE: tpconvmb reuses rcvbuf for output and will return
     iolen bytes that were converted to UTF-8. To correctly
     output this as a string we need to add a NULL terminator
     to rcvbuf or use another char* and strncpy to it.
     *********************************************************
```

```
*/
     *(rcvbuf + iolen) = '\0';
     /*output received buf from TOUPPERMB service converted to
     UTF-8 */ (void) fprintf(stdout, "simpclmb output string
     is: %s, Length %d\n", rcvbuf, iolen);

     /* Free Buffers & Detach from System/T */
     tpfree(sendbuf);
     tpfree(rcvbuf);
     tpterm();
     return(0);
}
```

Server-Side Application

```
/* #ident "@(#)apps:simpapp/simpservmb.c 1.0" */

#include <stdio.h>
#include <ctype.h>
#include <atmi.h> /* TUXEDO Header File */
#include <userlog.h> /* TUXEDO Header File */

/* tpsvrinit is executed when a server is booted, before it
begins processing requests. It is not necessary to have this
function. Also available is tpsvrdone (not used in this
example), which is called at server shutdown time.
*/

#if defined(__STDC__) || defined(__cplusplus)
tpsvrinit(int argc, char *argv[])
#else
tpsvrinit(argc, argv)
int argc;
char **argv;
#endif
{
     int ret,iolen;

  /* userlog writes to the central TUXEDO message log */
     userlog("Welcome to the simpservmb server");

  /* Some compilers warn if argc and argv aren't used. */
     argc = argc;
     argv = argv;
/*

     *********************************************************
     If it is desired to have automatic multibyte conversion
     "OFF" then comment out, or delete, the following six
     lines.
     The tuxsetmbaconv will only control this client process.
     The server process will need to set it's own environment
     variable or execute it's own tuxsetmbaconv() function.
     NOTE: An alternative to using these two lines is to set
     the TPMBACONV environment variable(eg. export
     TPMBACONV="YES").

     *********************************************************
*/
     ret = tuxsetmbaconv(MBAUTOCONVERSION_ON,0);
     if(ret == -1) {
         (void) fprintf(stderr, "tuxsetmbaconv failed\n");
         exit(1);
     }
       userlog("tuxsetmbaconv ON done");
/*

     *********************************************************
     NOTE:An alternative to using the following six lines is
     to set the TPMBENC environment variable(eg export
     TPMBENC="UTF-16BE").
     *********************************************************
*/
     ret = tuxsetmbenc("UTF-16BE",0);
     if(ret == -1) {
          (void) fprintf(stderr, "tuxsetmbenc failed\n");
          exit(1);
     }
          userlog("tuxsetmbenc UTF-16LE done");
          return(0);
}

/* This function performs the actual service requested by the
client. Its argument is a structure containing among other
things a pointer to the data buffer, and the length of the data
buffer.
*/
#ifdef __cplusplus
extern "C"
#endif
void
#if defined(__STDC__) || defined(__cplusplus)
TOUPPERMB(TPSVCINFO *rqst)
#else
TOUPPERMB(rqst)
TPSVCINFO *rqst;
#endif
{
     int i,ret;
     char myenc[80];
     char *en=&myenc[0];


     userlog("TOUPPERMB Input Length: %d", rqst->len);
     /*
     *********************************************************
     The automatic conversion is turned on, see the tpsvrinit
     above,and the server process will have already converted
     the buffer to the defined encoding before delivering it to
     this service. The rqst data should now be in the UTF-16BE
     encoding and the rqst length would now be twice what it
     was in the UTF-8 encoding
     *********************************************************
     */
     ret = tpgetmbenc(rqst->data,en,0);
     if(ret == -1) {
          (void) fprintf(stderr, "tpgetmbenc failed.\n");
          (void) fprintf(stderr, "Tperrno = %d\n", tperrno);
          tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
     }
     if(strcmp(en,"UTF-16BE") !=0) {
          (void) fprintf(stderr, "tpgetmbenc not==UTF-
       16BE.Got: %s\n",en);
          (void) fprintf(stderr, "Tperrno = %d\n", tperrno);
          tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
     }
       userlog("tpgetmbenc check==UTF-16LE done");
```

```
/*
     ********************************************************
     If it is desired to have automatic multibyte conversion
     "OFF" but to do the conversion on demand in this
     application then use the following 12 lines of code as an
     example. If no automatic conversion is done and tpconvmb
     is not executed then the rqst data bytes will be left
     defined by the same encoding name as the client process(ie
     UTF-8).
     ********************************************************

     if(tuxgetmbaconv(0) == MBAUTOCONVERSION_OFF) {
            ret = tpconvmb(&rqst->data, &iolen, "UTF-16BE",
     (long)0);
            if(ret == -1) {
                (void) fprintf(stderr, "Can't execute
     tpconvmb.\n");
                (void) fprintf(stderr, "Tperrno = %d\n",
      tperrno);
     tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
     }
     userlog("tpconvmb new mbstring length: %d",iolen);
     }
*/
     for(i = 0; i < rqst->len; i++) {
          if(rqst->data[i]) {
          userlog("TOUPPERMB index: %d, char: %c", i, rqst-
          >data[i]);
          rqst->data[i] = toupper(rqst->data[i]);
          } else {
     /*

     ********************************************************
     NOTE: The danger of arbitrarily using string/print
     functions on the data received: The original data was sent
     in UTF-8 but the converted data received is now in UTF-
     16BE and will have embedded NULLs. The LIBC string/print
     functions would not work correctly or crash.
     ********************************************************
     */
     userlog("TOUPPERMB skip index: %d",i);
    }
}
/* Return the transformed buffer to the requestor. */
tpreturn(TPSUCCESS, 0, rqst->data, rqst->len, 0);
}
```

Using FLD_MBSTRING

The final example illustrates the use of FLD_MBSTRIN.

Client-Side Application

```
#include <stdio.h>
#include <stdlib.h>
#include <atmi.h>
#include <userlog.h>
#include <fml.h>
#include <fml32.h>
#include "fmltbl32.h"
/*
  ****************************************************
  The fmltbl32 header file used to generate fmltbl32.h
  is simply a single field definition:
  # name    number  type     flags   comments
  FLD4   112     mbstring -       -
  ****************************************************
*/
#define BUFLEN 1024
#ifdef _TMPROTOTYPES
main(int argc, char *argv[])
#else
main(argc, argv)
int     argc;
char    *argv[];
#endif
{
    FBFR32*fmlptr;
    long rlen;
    int ret;
    char *fldmbio;
    FLDLEN32 packedlen;
/*
  **********************************************************
  This example sets two occurences of the same field to UTF-8
  packed data and then sends it to the FML32SRV service. The
  service will return the buffer with its fields in UTF-16BE
  format which will locally be converted back to UTF-8.
  **********************************************************
*/
    /* Attach to System/T as a Client Process  */
    if (tpinit((TPINIT *)NULL) == -1) {
    (void) fprintf(stderr,"tpinit failed: %s\n",
tpstrerror(tperrno));
    exit(1);
    }
    ret = tuxsetmbaconv(MBAUTOCONVERSION_ON,0);
    if(ret == -1) {
(void) fprintf(stderr, "tuxsetmbaconv failed\n");
    exit(1);

    }

(void) fprintf(stderr, "tuxsetmbaconv ON done.\n");
/*
  **********************************************************
  Since automatic conversion is turned on we need to set the
  encoding for the process environment. This is so that the
  reply to the tpcall will be converted back to UTF-8 before
  being made available to this application code.
  **********************************************************
*/
    ret = tuxsetmbenc("UTF-8",0);
    if(ret == -1) {
     (void) fprintf(stderr, "tuxsetmbenc failed\n");
     exit(1);
    }
    (void) fprintf(stderr, "tuxsetmbenc UTF-8 done.\n");
    /* allocation for fml32 buffer */
    if ( (fmlptr = (FBFR32 *) tpalloc("FML32", NULL, BUFLEN) ==
NULL ) {
     (void) fprintf(stderr,"tpalloc failed: %s\n",
     tpstrerror(tperrno));
     tpterm();
     exit(1);
    }


     /* create and pack datastream input for FLD_MBSTRING
     fields */ packedlen = 256;/*excessive space, actual bytes
     used is very little*/ fldmbio =
     (char*)malloc((size_t)packedlen);
     if ( Fmbpack32("UTF-8", "hello", 5, fldmbio,
     &packedlen,0) < 0 ) {
          (void) fprintf(stderr,"Fmbpack32 on hello failed:
          %d\n", Ferror32);
          exit(1);
    }
    /*set 1st occurence of FLD_MBSTRING field FLD4*/
    if ( Fchg32(fmlptr, FLD4, (FLDOCC32)-1, fldmbio, packedlen)
< 0 ) {
          (void) fprintf(stderr,"Fchg on FLD4,0 failed: %d\n",
Ferror32);
          exit(1);
    }
    userlog("Fchg on FLD4,0 passed. packedlen: %d", packedlen);


    packedlen = 256;
    if ( Fmbpack32("UTF-8", "world", 5, fldmbio, &packedlen,0)
< 0 ) {
          (void) fprintf(stderr,"Fmbpack32 on bobf failed:
%d\n", Ferror32);
          exit(1);
    }
    /*set 2nd occurence of mbstring field FLD4*/
    if ( Fchg32(fmlptr, FLD4, (FLDOCC32)-1, fldmbio, packedlen)
< 0 ) {
          (void) fprintf(stderr,"Fchg on FLD4,1 failed: %d\n",
Ferror32);
          exit(1);
    }
    userlog("Fchg on FLD4,1 passed. packedlen: %d", packedlen);
/*
  **********************************************************
  Note: Since all fields are defined using the same encoding,
  an alternative to setting each encoding separately would be
  to use tpsetmbenc(UTF-8) on the FML32 buffer and then use
  Fmbpack32() with FBUFENC for the flag arg and NULL for the
  encoding arg. This would reduce the total size of the buffer
  used.
  **********************************************************
*/

    puts("The FML32 buffer sent : -");

    Fprint32(fmlptr);
    userlog("Fchg32 : successful");
    /*send the FML32 buffer to the FMLSRV32 service*/
     if(tpcall("FMLSRV32",(char*)fmlptr,0,(char**)&fmlptr,&rle
      n,TPNOTIME) == - 1 ) {
           (void) fprintf(stderr,"tpcall failed: %s\n",
           tpstrerror(tperrno));
          exit(1);
    }

    puts("The FML32 buffer got : -");
    Fprint32(fmlptr);

    tpfree((char *)fmlptr);
    tpterm();
    exit(0);
}
```

Server-Side Application

```
#include <stdio.h>
#include <ctype.h>
#include <atmi.h>    /* TUXEDO Header File */
#include <userlog.h>    /* TUXEDO Header File */
#include <fml.h>
#include <fml32.h>
#include "fmltbl32.h"

/* tpsvrinit is executed when a server is booted, before it
begins processing requests. It is not necessary to have this
function. Also available is tpsvrdone (not used in this
example), which is called at server shutdown time.
*/

#if defined(__STDC__) || defined(__cplusplus)
tpsvrinit(int argc, char *argv[])
#else
tpsvrinit(argc, argv)
int argc;
char **argv;
#endif
{
    int ret=0;
    /* Some compilers warn if argc and argv aren't used. */
    argc = argc;
    argv = argv;

    /* userlog writes to the central TUXEDO message log */
    userlog("Welcome to the simple server");

    ret = tuxsetmbaconv(MBAUTOCONVERSION_ON,0);
    if(ret == -1) {
        (void) fprintf(stderr, "tuxsetmbaconv failed\n");
        exit(1);
    }
    userlog("tuxsetmbaconv ON done");

    ret = tuxsetmbenc("UTF-16BE",0);
    if(ret == -1) {
     (void) fprintf(stderr, "tuxsetmbenc failed\n");
     exit(1);
    }
    userlog("tuxsetmbenc UTF-16BE done");

    return(0);
}

/* This function performs the actual service requested by the
client. Its argument is a structure containing among other
```

```
things a pointer to the data buffer, and the length of the data
buffer.
*/

#ifdef __cplusplus
extern "C"
#endif
void
#if defined(__STDC__) || defined(__cplusplus)
FMLSRV32(TPSVCINFO *rqst)
#else
FMLSRV32(rqst)
TPSVCINFO *rqst;
#endif
{
    char buf[1024];
    char odata[1024];
    char pckdata[1024];
    char encname[256];
    char *bufptr = (char *)(rqst->data);
    int i=0,occ=0;
    FLDLEN32 odatalen=0,packedlen=0,buflen=0;
    userlog("Welcome to the fml32srv server");
/*
  **********************************************************
  Since automatic conversion is turned on the FML32 buffer
  that this FMLSRV32 service will receive will have been
  converted to the local encoding(ie UTF-16BE). The following
  code will get the fields from the fml32 buffer, extract the
  userdata from the fields, manipulate the date(ie change to
  uppercase), repack it, change the fields and then send the
  fml32 buffer back to the client.
  **********************************************************
*/


     for (occ = 0;occ < 2; occ++) {
        buflen = 1024;
          /*get FLD_MBSTRING field from FML32 buffer*/
        if ( Fget32((FBFR32 *)bufptr, FLD4, occ, buf, &buflen)
           == -1 ) { userlog ("Fget32 FLD4,%d failed: %d", occ,
           Ferror32); tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
        }
        userlog("FMLSRV32 Fget32 FLD4,%d passed buflen: %d",
           occ, buflen);
        odatalen = 1024;
           /*unpack the field into user data and encoding
           info*/
        if ( Fmbunpack32(buf, 20, encname,odata,&odatalen,0) ==
           -1 ) { userlog ("Fmbunpack32 FLD4,%d failed", occ);
            tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
        }
        userlog("FMLSRV32 FLD4,%d encname: %s", occ, encname);
             /*change relevant bytes to uppercase*/
       for(i = 0; i < odatalen; i++) {
            if(odata[i]) {
             userlog("FMLSRV32 FLD4,%d index: %d, char:
             %c",occ,i,odata[i]); odata[i] = toupper(odata[i]);
            } else {
             userlog("FMLSRV32 FLD4,%d skip index: %d", occ,
             i);
            }
        }
        packedlen = 1024;
```

```
          /*pack encoding name and user data into field data*/
        if ( Fmbpack32("UTF-
        16BE",odata,odatalen,pckdata,&packedlen,0) < 0 ) {
          userlog("Fmbpack32 on FLD4,%d failed: %d", occ,
          Ferror32);
          tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
        }
          /*set the FLD_MBSTRING with new packed data*/
        if (Fchg32((FBFR32
        *)bufptr,FLD4,(FLDOCC32)occ,pckdata,packedlen) < 0) {
          userlog("Fchg32 on FLD4,%d failed: %d", occ,
          Ferror32);
          tpreturn(TPFAIL, 0, rqst->data, 0L, 0);
        }
        userlog("Fchg32 on FLD4,%d passed. packedlen: %d ",
        occ, packedlen);
    }
    userlog("Successfully done with the fml32srv server");

    /* Return the transformed buffer to the requestor. */
    tpreturn(TPSUCCESS, 0, rqst->data, 0L, 0);
}
```

Custom Conversion Function

```
/*
 * CUSTmbconv
 *
 * This function will convert characters from a source
encoding, defined in the TCM, to a targe codest.
 *
 * INPUT
 *      *iptr        - pointer to an input buffer
 *      ilen         - Length of input buffer
 *      *target_enc    - Characters in iptr will get converted
                     to the code set defined by this name.
 *      *flags      - valid values are TMUSEIPTR or TMUSEOPTR.
Where results put.
 *
 * OUTPUT
 *      *optr      - pointer to an output buffer. If null, use
iptr to output.
 *      olen       - Length of output buffer. If optr is null,
use ilen.
 *      *flags     - valid values are TMUSEIPTR or TMUSEOPTR.
Where results put.
 *
 * RETURNS
 *      -1                - Failure (check errno for reason)
 *      positive #        - Success. Return val is num of bytes
used in result.
 *      negative #        - Not enough space. Value is -1*(guess
of bytes needed)
 */
/*ARGSUSED*/
long
#ifdef _TMPROTOTYPES
_TMDLLENTRY
CUSTmbconv(char _TM_FAR *iptr, long ilen, char _TM_FAR
*target_enc, char _TM_FAR
     *optr, long olen, long _TM_FAR *flags)
#else

CUSTmbconv(iptr, ilen, target_enc, optr, olen, flags)
char *iptr;
```

```
long ilen;
char *target_enc;
char *optr;
long olen;
long *flags;
#endif
{
    iconv_t cd;
    char    *tptr;
    char    *to = (char *)NULL;
    char  *fptr;
    size_t  ileft, oleft, ret,used=0;
    char  encname[56];
    char  *src_enc = &encname[0];

    if ( (target_enc == NULL) || (*target_enc == '\0') ) {
        /* missing target encoding argument */;
        return(-1);/*WILL NEED TO SET TPERRNO for return -1*/
    }

    if(tpgetmbenc(iptr,src_enc,0) < 0) {
        /* missing source encoding name */;
        return(-1);
    }

    /* convert characters from source encoding to target
    encoding format */
    cd = iconv_open((const char *)target_enc, (const char
    *)src_enc);
    if (cd == (iconv_t)-1) {
        /* iconv_open failure */
        return (-1);
    }

    if(optr == NULL) {
        /* If no output buf given and if conversion fails due
        to insufficient*/
        /* buf size then the input buf would be unusable when
        sent back for  */
        /* a retry attempt. So use tmp buffer for output until
        conversion is */
        /* clean and copy it back to the input buffer upon
        successful conv*/
        if(olen == 0) {
           /*probably will throw E2BIG error with correct size
           to use*/
            olen = ilen;
        }
        /*if olen!=0 then it should be the max size of the iptr
buffer*/
        if ((to = (char *)malloc((size_t)olen)) == NULL) {
           return (-1);
        }

    } else {
        to = optr;
    }
    tptr = to;
    fptr = iptr;
    ileft = ilen;
    oleft = olen;

    (void) iconv(cd,NULL,NULL,NULL,NULL); /* go to the initial
state */
    for ( ;; ) {
        ret = iconv(cd, &fptr, &ileft, &tptr, &oleft);
        if (ret != (size_t)-1) {
            /* iconv succeeded. NOTE: Some characters may not
have needed */
          /* conversion and are the same as input value
          representation */
          used = used + (olen - oleft);
          olen = oleft;
          if(ileft != 0) {
               /* iconv not done, execute again*/
               continue;
          }
          if(optr == NULL) {
                /* no output buffer given,copy tptr buf back to
                input buffer*/
                (void) memcpy(iptr,to,used);/*DANGER:iptr len
                 must be >= used*/
                free(to);
                *flags |= TMUSEIPTR;
            } else {
                /* characters from iconv exec in output buffer,
optr */
                *flags |= TMUSEOPTR;
            }
             (void) iconv(cd,NULL,NULL,NULL,NULL);/* reset to
             initial state */
            (void) iconv_close(cd);
            return (used);/*return #bytes used in output
             buffer*/
        } else {
            /* iconv failed */
            (void) iconv(cd,NULL,NULL,NULL,NULL);/* reset to
             initial state */
            (void) iconv_close(cd);
            if(optr == NULL) {
                free(to);
            }
            if (errno == E2BIG) {
                olen = (ilen + (4 * ileft)) * -1;
                *flags |= TMUSEIPTR;
                return (olen);/*return guesstimate of size iptr
                should be*/
            } else if (errno == EINVAL) {
                /* Incomplete char/shift sequence */
            } else if (errno == EILSEQ) {
                /* NOTE: We do not handle code set state
                dependent sequences */
            } else if (errno == EBADF) {
                    /* Actually, this should happen above
                    during iconv_open */
            } else {
                /* Undefined error */
            }
            return(-1);
        }
    }
}
```

Oracle Tuxedo Globalization Features: Multibyte Support for the Asia Pacific Region
Updated June 2008

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com