BEA Tuxedo のグローバル化機能

アジア太平洋地域向けの
マルチバイトサポート

BEA ホワイトペーパー

著作権

2003 年 2 月 15 日

限定的権利条項

商標

目次

Tuxedo におけるグローバル化の戦略と機能

Tuxedo リリース 4.1 以降、インターナショナライゼーションとローカライゼーションは同システムの不可欠な要素となっています。これに対して、BEA Tuxedo リリース 8.1 では、アジア太平洋地域ロケールでのマルチバイト文字コードセット処理をフルサポートしています。開発者は、BEA Tuxedo リリース 8.1 を使用してソリューションを構築する際には、もはや言語上の制限を受けることも、インターナショナライゼーションを実現するためにソフトウェアをカスタマイズする必要もありません。さらに、BEA Tuxedo 8.1 は、BEA Tuxedo のそれ以前のリリースとの完全な互換性と相互運用性を備えています。

BEA Tuxedo 8.1 では、インターナショナライゼーションに関して、以下のような機能拡張が施されています。

・ユーザ データ用のマルチバイト文字型バッファをサポート
・日本語、中国語、および韓国語のコードセットエンコーディングの間で、API を使ったプログラムによるオンデマンド変換、または自動変換が可能
・コードセットエンコーディング情報の「取得」と「設定」、およびプログラムまたはコンフィグレーション設定による自動変換のオン/オフ切り替えが可能
・変換ライブラリをカスタム変換機能に容易に差し替え可能

これらの Tuxedo 8.1 固有の拡張機能では、MBSTRING という新しい型付きバッファ、FLD_MBSTRING という新しいフィールド型、マルチバイト文字転送および変換用の新しい API など、いくつかの新しいシステム機能が利用されます。

BEA Tuxedo 8.1 では、プログラマは、エンコーディング変換の管理を、環境変数 TPMBENC および TPMBACONV を使ってコンフィグレーション設定で行うこともできますし、新しい API 関数を使ってプログラムで行うこともできます。自動変換のオン/オフをプログラムで切り替えることができるため、アプリケーションでは必要なときにだけ変換が行われるように制限できるようになり、変換に関わるパフォーマンスをうまく管理することができます。

自動コードセットエンコーディング変換を行うように BEA Tuxedo 8.1 システムがコンフィグレーションされている場合には、異なるコンピュータプラットフォーム上で動作するプロセス間で MBSTRING バッファ (または、FML32 内の FLD_MBSTRING フィールド) が転送されると、Tuxedo システムによって裏でコードセットエンコーディング間の変換が行われます。特に、受信側が MBSTRING バッファを送信側のコードセットエンコーディング表現から受信側のコードセットエンコーディングに自動的に変換します。一方、環境変数 TPMBENC および TPMBACONV を使って手動で自動コードセット変換がコンフィグレーションされていない場合、送信側または受信側のアプリケーションで、変換用 API (詳細については「付録 1」を参照) を使ってケースバイケースでコードセットエンコーディング変換を要求することができます。

GNU の変換ライブラリ inconv を使用すると、Unix プラットフォームと Windows プラットフォームで共通にコードセット変換を行えるようになります。BEA Tuxedo の型付きバッファを使用すれば、たとえばテストやパフォーマンスチューニングのために、変換ライブラリをカスタム機能にたやすく差し替えることができます。

このホワイトペーパでは、BEA Tuxedo 8.1 のインターナショナライゼーション機能について説明し、例を用いて新しい機能を紹介します。

概観

オペレーティング システム、ライブラリ、開発ツールなどの大半のソフトウェア製品は、国際的な環境向けに設計され開発されます。たとえば、東京に本社がありニューヨークとソウルに支社があるような大企業では、英語、日本語、および韓国語のソフトウェア環境を組み合わせなければならないことがあります。そうしたソリューションでは、言語面、文化面、およびプレゼンテーション面での要件が非常に異なる環境で、同じソフトウェアが動作する必要があります。さらに、このような国際的に分散したコンピューティング環境では、数値、時刻、日付、通貨表示形式、メッセージ表現、およびコードセットエンコーディング体系の地域ごとの変動もサポートしなければなりません。トランザクションが世界中のさまざまな地域に及ぶにつれて、これらの要件をすべて(アプリケーションを再起動せずに)「オンザフライ」でサポートする必要があります。こうした要件を満たすソフトウェアは、「グローバル化された」ソフトウェアと呼ばれます。

グローバル化

ソフトウェアのグローバル化は、「インターナショナライゼーション」と「ローカライゼーション」の双方の要件に対処することで達成されます。インターナショナライゼーションによって、ソフトウェアは言語と習慣の異なる地域間で共通に使用できるようになります。インターナショナライズされたソフトウェアを作成するには、開発者はまず、言語と文化に依存するプログラム部分を切り分けます。たとえば、エラーメッセージは、その対象となる「ロケール」の言語に翻訳しやすいように分離されます。ロケールとは、同じ言語と習慣を共有する地理的または政治的な地域のことです。インターナショナライズされるプログラムは、システムの初期化時にロケール依存部分を取得するように設計または修正されます。

ローカライゼーションは、ロケール依存部分のロケール固有バージョンすなわち「パッケージ」を作成するプロセスのことです。ローカライゼーションには、ユーザインタフェースに表示されるラベル、エラーメッセージ、オンラインヘルプなどのテキストを翻訳する作業が含まれます。また、通貨の値、時刻、日付、数値などのデータ項目を文化固有の形式に書式化する作業も必要です。BEA では、顧客の必要に応じて、利用可能なローカライゼーションパッケージを開発します。

コードセットとエンコーディング

「文字セット」とは、所定の言語でテキストを表す要素の集合のことです。英語のアルファベットは文字セットの一例です。文字間に暗黙の順序関係が存在する場合もありますが、文字には特定の値は割り当てられません。たとえば、英語のアルファベットを順に読み上げる場合、「A, B, C, ...」で始めて「..., X, Y, Z」で終わるまで、習慣に従って読み上げを続けることになるでしょう。このような暗黙の順序付けはありますが、その関係を暗黙に意味するような数値関係は文字間にはありません。「コードセット」は、そのような数値関係を提供するもので、コンピュータプログラムで文字セットを操作するためのメカニズムを与えてくれます。

コードセットは「コード化文字セット」とも呼ばれ、コンピュータでの利用を前提に、文字を負でない一意な整数にマッピングしたものです。あるコードセットについての一意なバイナリ値のマッピングは、そのコードセットの「エンコーディング」と呼ばれます。米国では、大半のコンピュータキーボードに刻字されている文字の集合を表すコードセットは、ASCII と UNICODE の 2 つです。ASCII はまた、エンコーディングでもあります。個々のコードセットに複数のエンコーディングが存在することもあります。たとえば、日本のコンピュータベンダでは、日本語のコードセットである Kanji 用に、EUC-JP、Shift-JIS (SJIS)、ISO-2022-JP の少なくとも 3 通りのエンコーディングをサポートしています。Unix ベンダでは EUC-JP をサポートしているものが大半で、SJIS もサポートしているベンダも一部にはあります。Windows、OS/2、および Macintosh では、SJIS をサポートしています。韓国では KSC5601 エンコーディングが広く利用されているのに対して、中国では GBK が使用されています。Java では、ネイティブの UNICODE と多数の他のエンコーディングをサポートしています。

マルチバイトエンコーディング変換の過去と現在

標準英語を含むヨーロッパ言語のアルファベット文字は、8 ビット (シングルバイト) のコードセットエンコーディング体系で対応できます。しかし、中国語、日本語、および韓国語は多数の記号すなわち「表意文字」で成り立っており、マルチバイトのコードセットエンコーディング体系が必要になります。BEA Tuxedo 8.1 では、マルチバイトコードセット処理機能で、これらのアジア太平洋地域用文字セットをサポートします。

BEA Tuxedo 8.1 が登場する前は、アプリケーション開発者は、グローバル化機能を実現するのにカスタム変換ソリューションを作成する必要がありました。しかし、カスタム変換で処理できるのは、非常に特化したケースだけです。たとえば、SJIS から EUC-JP への変換を処理するカスタムソリューションがあっても、SJIS と ISO-2022-JP の間の変換を行うには、別のソリューションが必要になるでしょう。BEA Tuxedo 8.1 がリリースされたことで、こうしたカスタム変換を開発する必要はもうありません。

なお、BEA Tuxedo のコードセット変換機能は、コードセットのエンコーディング (たとえば、UNICODE コードセットのエンコーディング UTF-8 と UTF-16BE) 間の変換を行うためのものであることを忘れないでください。決して、コードセット (たとえば、ASCII と UNICODE) 間の変換でも言語間の翻訳でも「なく」、あくまで同一言語の異なるエンコーディング間の変換です。

典型的な変換シナリオ

別個のプラットフォーム (たとえば、クライアントシステムとサーバシステム) 上で異なる Kanji エンコーディング体系が使用されるというのが、マルチバイト変換の非常によくあるシナリオです。異なるエンコーディング体系を使用するプラットフォーム上でクライアントとサーバが動作する場合、プラットフォーム間でエンコーディング変換を実行する必要があります。図 1 の例では、日本における分散コンピューティング環境でのシナリオを示しています。この例では、クライアントは、SJIS をサポートする Windows マシン上で動作します。BEA Tuxedo サーバは Unix マシンで、EUC-JP をサポートしています。

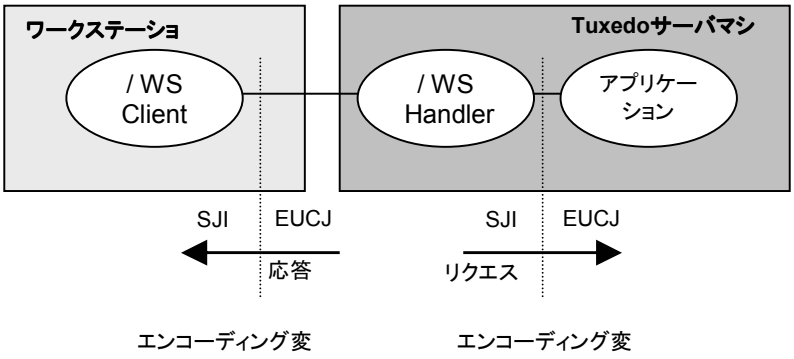

図 1: 典型的なシナリオ

BEA Tuxedo のマルチバイト変換機能

上記の例では、MBSTRING というバッファ・タイプを使って、マルチバイト文字のアプリケーションデータが Windows クライアントと BEA Tuxedo サーバプロセスの間で転送されます。このバッファ・タイプに関連付けられている API 関数群によって、コードセットエンコーディング名の「取得」や「設定」が行われ、GNU iconv ライブラリを使ってマルチバイトデータ変換が実行されます。

MBSTRING バッファは、バッファ・タイプスイッチ構造体 (tm_typesw) に新しいエントリを追加することで、BEA Tuxedo に実装されています。BEA Tuxedo では、バッファ・タイプごとにどのルーチンを呼び出すかを、この構造体によって決定することができます。MBSTRING バッファの場合、Tuxedo システムは内部関数_mbsconv()を呼び出して、コードセットの自動マルチバイトデータ変換を実行します。この内部関数では、GNU ライブラリルーチンを使用して、ユーザデータを変換します。BEA Tuxedo システムに定義されている「MBSTRING」でのバッファ・タイプの使い方の詳細については、 buffer(3)、typesw(5)、および tuxtypes(5)の各マニュアルページを参照してください。

エンコーディング変換の管理

エンコーディング変換の管理には、環境変数 TPMBENC および TPMBACONV を使ってコンフィグレーションの設定として行う方法と、API 関数を使ってプログラム内で行う方法の 2 通りがあります。自動変換を行うように環境変数が設定されている場合には、受信側の BEA Tuxedo システムがバッファ内のデータを別のエンコーディングに変換します。そうでない場合には、プログラミングインタフェース tuxsetmbaconv()を使用することで、アプリケーションを再起動せずに自動エンコーディング変換のオン/オフを切り替えることができるため、必要なときにだけ変換が行われるように制限できるようになります。そうでないと、ホップごとに変換が行われるおそれがあり、パフォーマンスに悪影響が出ます。

図 2 に示すのは図 1 で説明したのと同じ例ですが、ここではさらに、BEA Tuxedo でのマルチバイトデータの処理方法を詳しく示しています。環境変数 TPMBENC および TPMBACONV がマシンごとに設定され、エンコーディングと自動エンコーディング変換の状態 (オンかオフ) を識別します。この例 (日本語ロケールの場合) では、SJIS エンコーディングをサポートする Windows クライアントと、EUC-JP エンコーディングをサポートする Unix サーバを示しています。型付きバッファのヘッダから、そのバッファが MBSTRING 型であることが特定されると共に、エンコーディングとデータ長の情報が提供されます。バッファそのものには、ヘッダで識別されたエンコーディングで表現されたユーザデータが格納されています。クライアントリクエストのバッファには、SJIS エンコーディングで表現されたデータが格納され、サーバ応答のバッファには EUC-JP エンコーディングで表現されたデータが格納されます。

アプリケーションの設計時には、考慮すべき点が 2 つあります。第 1 に、変換は本来、パフォーマンス的にコストがかかります。図の例では、自動変換を使用する場合、1 つのメッセージにつき変換が 2 回行われることになります (リクエストがサーバに受信されたときに 1 回と、応答がクライアントに受信されたときにもう 1 回)。 第 2 に、バッファ内のユーザデータのサイズは変換によって変わります。この例の場合、クライアント側では、バッファは
変換後も同じサイズになるか、小さくなるかのどちらかです。サーバ側では、バッファサイズは同じままか大きくなります。

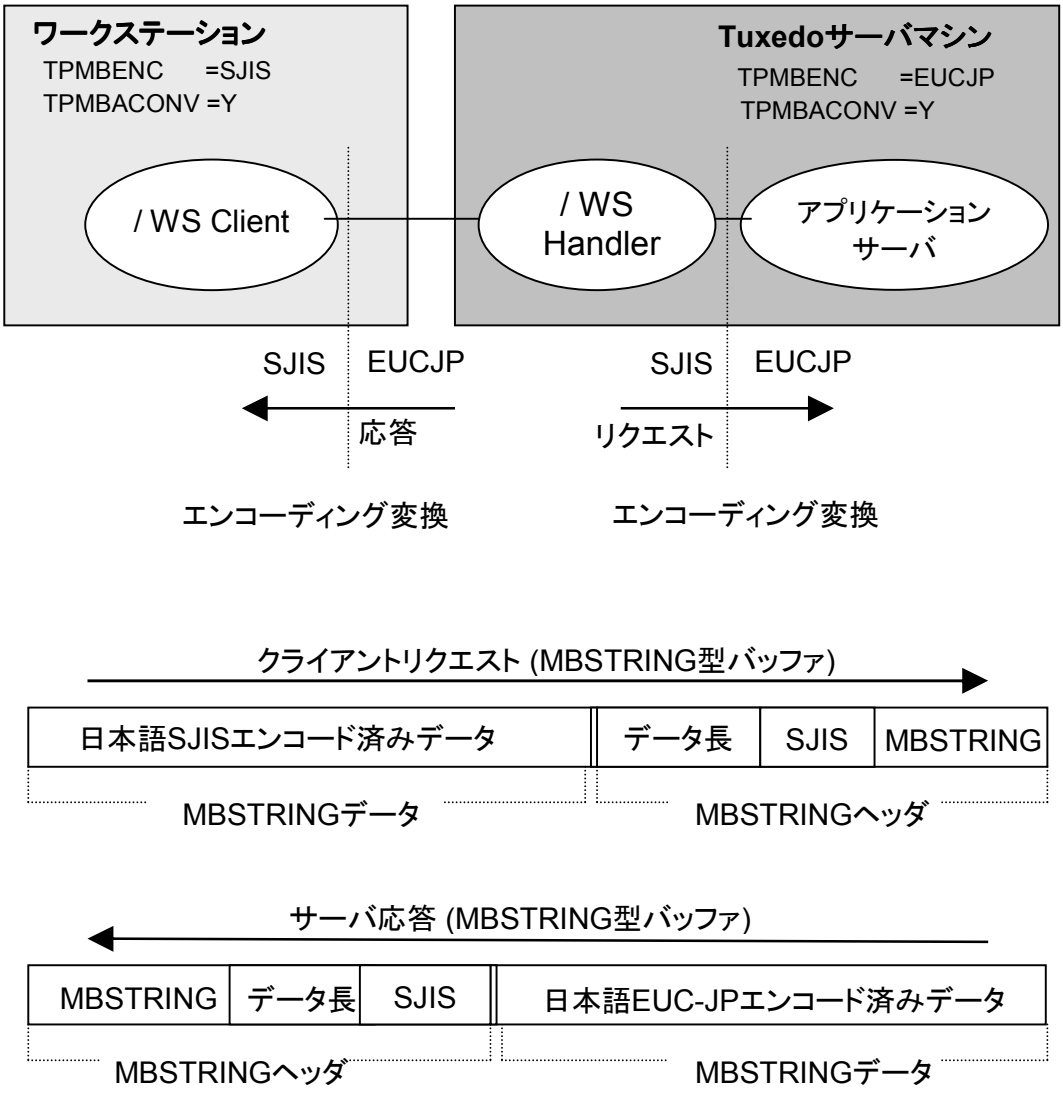

図 2: MBSTRING 型バッファを使ったデータ変換

クライアントサイドでの処理

クライアントプロセスが呼び出されると、その動作マシンでサポートされているコードセットエンコーディングの名前が取得または設定されます。たとえば、環境変数 TPMBENC を使って設定されたエンコーディング名を取得するために、クライアントでは tuxgetmbenc()を呼び出して、

環境リスト内で「TPMBENC=<エンコーディング名>」の形式の文字列を探します。その文字列が存在する場合には、クライアントが BEA Tuxedo の tpalloc()を呼び出して新しい MBSTRING バッファを割り当てる際に、エンコーディング名がユーザデータと共に渡されます。そのエンコーディング名はそのあとキャッシュに入れられるため、tpalloc()の呼び出しは、プロセスで初めてタイプ・スイッチ関数が呼び出されるときに 1 回だけ行えば済みます。環境変数 TPMBENC が未定義の場合、あるいは処理中にリセットしたい場合には、アプリケーション側で API 関数を使用して定義することができます。

クライアントからいったん tpalloc()が呼び出されると、図 3 に示したように、バッファ割り当てとデータ変換が BEA Tuxedo 側で行われます。BEA Tuxedo システムの内部では、新しい MBSTRING バッファ用のメモリを割り当て、内部用 tuxgetmbenc()関数を使用して、TPMBENC 環境変数に定義されているエンコーディング名を取得します (その環境変数が設定されている場合)。BEA Tuxedo は、そのエンコーディング名を MBSTRING バッファのヘッダに追加し、割り当てたバッファをクライアントに返します。

あとでクライアントが、たとえば tpsend()や tpcall()を使って MBSTRING バッファを送信すると、BEA Tuxedo は再び介入して、受信側での変換を実行します。これについては、このあとの「サーバサイドでの処理」の節で説明します。

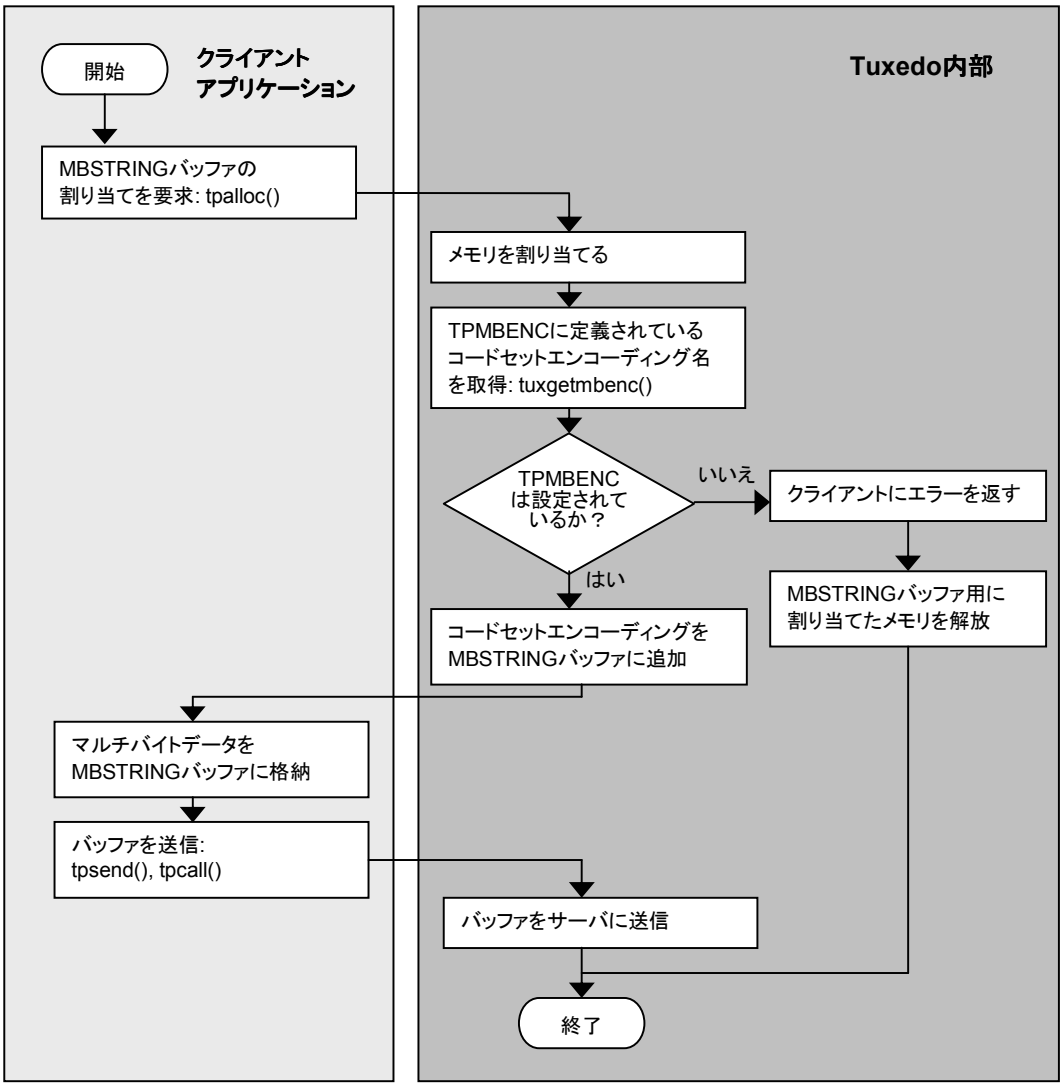

図 3: クライアント側の処理

サーバサイドでの処理

図 4 のフローチャートでは、MBSTRING 型バッファの入ったリクエストがクライアントからサーバに送信されたときに内部で行われる BEA Tuxedo の処理を示しています。図 4 ではクライアントが応答を受信したときに行うのと同じ手順も示していることに注意してください。BEA Tuxedo では、メッセージをサービスに渡す前に、MBSTRING バッファを受信します。そして、環境変数 TPMBACONV を調べて、自動変換が設定されているかどうかを判断します。設定されていなければ、BEA Tuxedo は、エンコーディング変換を行わずに MBSTRING バッファ内

のデータをサーバに送ります。自動変換が設定されている場合、BEA Tuxedo は TPMBENC に定義されているエンコーディング名を取得します。エンコーディング値が設定されていないと変換を行えないので、何らかのエラーチェックを行って、値が設定されていることを確かめます。万一値が設定されていなければ、エラーをログに記録し、サーバに制御を渡すことになります。

TPMBENC 環境変数が設定されている場合、BEA Tuxedo のタイプ・スイッチエレメントがクライアントとサーバのエンコーディング名を自動的に比較し、双方のエンコーディング名が異なる場合、BEA Tuxedo は、GNU iconv ベースのライブラリルーチンまたはユーザ作成のカスタム変換ルーチンを使って、着信メッセージのエンコーディングを、サーバのマシンでサポートされているエンコーディングに自動的に変換します。カスタムルーチンの作成の詳細については、9 ページの「カスタマイズ」を参照してください。BEA Tuxedo は、変換したデータをサービスに送り、制御をそのサービスに渡します。サーバサイドおよびクライアントサイドのマルチバイト変換アプリケーションのサンプルについては、付録 2 を参照してください。

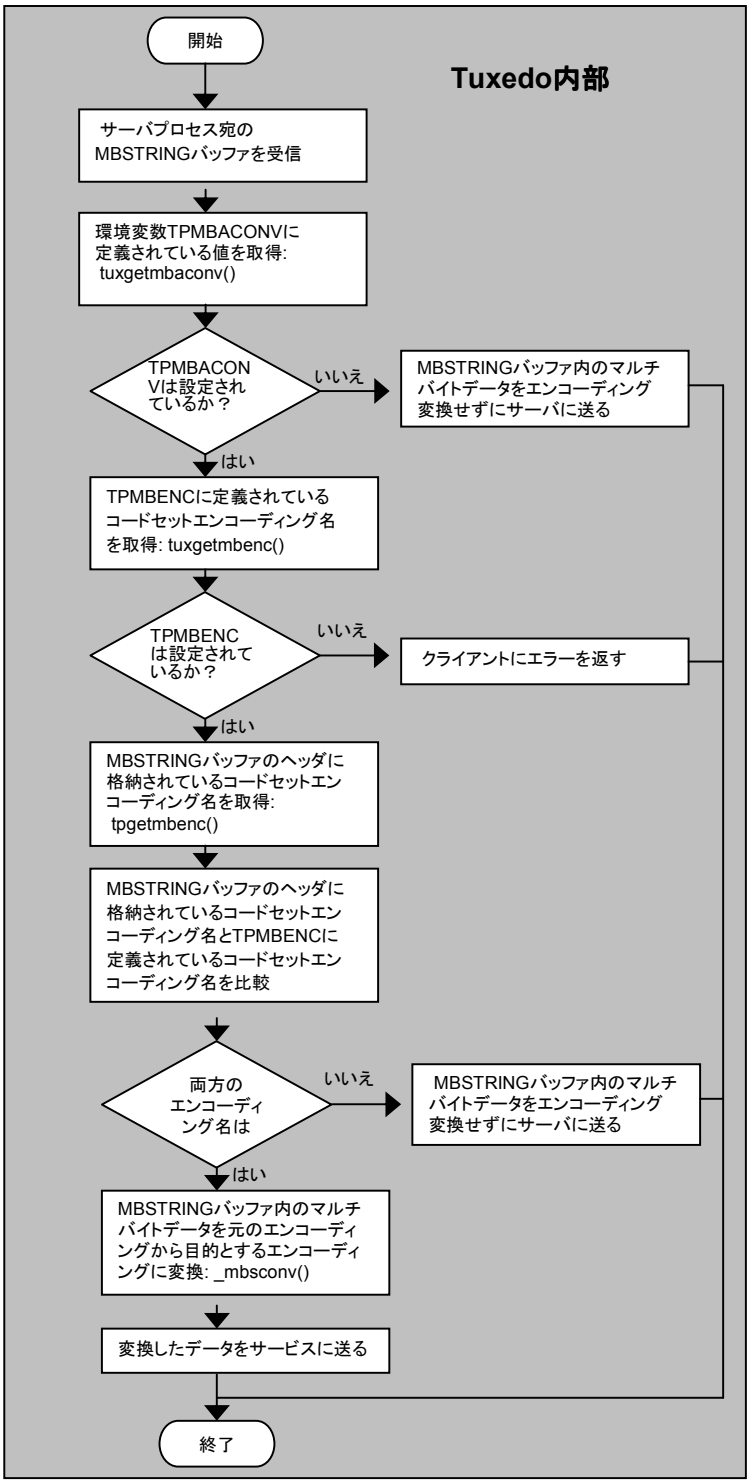
開始
Tuxedo内部
サーバプロセス宛の
MBSTRINGバッファを受信
環境変数TPMBACONVに
定義されている値を取得:
tuxgetmbaconv()
TPMBACON
Vは設定され
ているか？
いいえ
MBSTRINGバッファ内のマルチ
バイトデータをエンコーディング
変換せずにサーバに送る
はい
TPMBENCに定義されている
コードセットエンコーディング名
を取得: tuxgetmbenc()
TPMBENC
は設定されて
いるか？
いいえ
クライアントにエラーを返す
はい
MBSTRINGバッファのヘッダに
格納されているコードセットエン
コーディング名を取得:
tpgetmbenc()
MBSTRINGバッファのヘッダに
格納されているコードセットエン
コーディング名とTPMBENCに
定義されているコードセットエン
コーディング名を比較
両方の
エンコーディ
ング名は
いいえ
MBSTRINGバッファ内のマルチ
バイトデータをエンコーディング
変換せずにサーバに送る
はい
MBSTRINGバッファ内のマルチ
バイトデータを元のエンコーディ
ングから目的とするエンコーディ
ングに変換: _mbsconv()
変換したデータをサービスに送る
終了

図 4: サーバ側の処理

カスタマイズ

たとえば、変換関数の呼び出しの組み込みやデバッグのためにパフォーマンスの向上が必要な場合や、用意されているライブラリでは処理できないカスタム文字が必要な場合には、開発者はカスタム変換関数を作成することができます。MBSTRING の定義においてデフォルトの変換関数の名前をカスタム関数の名前に置き換えることで、MBSTRING のカスタム自動変換ルーチンをたやすくインストールできます。MBSTRING は tmtypesw.c ファイルで定義され、BEA Tuxedo のバッファ・タイプはこのファイルでプロセスバッファ・タイプスイッチ tm_typesw に追加されます。BEA Tuxedo でのバッファ・タイプ定義の詳細については、buffer(3) のマニュアルページを参照してください。アプリケーションレベルで tpconvmb()関数を使用すれば、自動変換機能とは無関係にバッファを変換することができます。

以下に示したのは tmtypesw.c ファイルの一部ですが、そこでは、MBSTRING のカスタム定義を記述しています。最後の行では、デフォルト変換関数の名前_mbsconv がカスタム関数の名前 CUSTmbconv に置き換えられていることを示しています。そのため、BEA Tuxedo で MBSTRING 型データの自動エンコーディング変換が実行される際には、デフォルト関数ではなく、カスタム変換ルーチンが呼び出されることになります。

```
"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       /* カスタマイズされたマルチバイトコードセット変換関数 */
```

CUSTmbconv のコードは、変換に通常使用される関数群から構成されますが、最終的には、Unix オペレーティングシステム上で通常使用される iconv 呼び出しに帰着されます。

カスタマイズ関数のサンプルは、付録 2 に掲載してあります。詳細については、typesw(5)のマニュアルページか、BEA Tuxedo 製品の配布キット内の$TUXDIR/lib/tmtypesw.c を参照してください。

エンコーディングエリアス名

GNU iconv の仕様では、「charset.alias」ファイルが使用できるようになっています。このファイルを使用することで、ユーザは既存のエンコーディング名のエリアスを定義することができます。この機能は、独自のエンコーディングの指定に一般に使用される名前 (SJIS、SHIFT_JIS、SHIFT-JIS、MS_KANJI、CSSHIFTJIS など) が定義されている GNU の組み込みリストと併せて利用できます。この機能は利用可能ではありますが、パフォーマンスコストがかかるので、あまり利用しないほうがよいでしょう。その代わり、GNU iconv の仕様で定義されている名前から 1 つ選んで、独自のエンコーディングに使用するようにしましょう。

FML バッファでのマルチバイトデータの取り扱い

BEA Tuxedo 8.1 のリリースによって、FML32 バッファでは、コードセット識別済みマルチバイトデータ用の FLD_MBSTRING フィールド型を使用できるようになりました。Fmbpack32()関数と Fmbunpack32()関数は、そのデータの処理に必要な情報をこのフィールドに格納します。「パックされた」データは FML32 バッファを使って送信され、TPMBACONV 環境変数が設定されている場合には、その FML32 バッファの受信側で FML32 バッファ・タイプスイッチ変換関数 _fmbconv32 が自動的に実行されます。この関数は、FML32 バッファをチェックして FLD_MBSTRING フィールドを調べ、そのフィールド情報に含まれているエンコーディング名がローカルの TPMBENC 環境変数値と同じでなければ、変換を実行します。_mbconv 関数の場合と同様に、ユーザは tmtypesw を再定義することで、カスタマイズを行うことができます。アプリケーションでは、FML32 API 関数と FLD_MBSTRING フィールド型を使って、FML32 バッファ内のパックされた変換済みデータにアクセスします。また、Fmbunpack32()関数を使ってデータをアンパックします。関連するこれらの関数の概要については、付録にまとめてあります。

FML32 バッファで MBSTRING と FLD_MBSTRING を使った例については、付録 2 を参照してください。

付録 1: マルチバイトデータに関連する API

表 1: MBSTRING 関連の関数

（表 1 の始まり）
tpconvmb()
入力バッファと一緒に渡されたエンコーディングから指定のエンコーディングに、文字を変換する。

tpgetmbenc()
tpsetmbenc()
クライアントプロセスまたはサーバプロセスで、MBSTRING バッファ内のコードセットエンコーディング名を取得したりリセットできるようにする。tpsetmbenc()は、エンコーディング名が設定されているかどうかを示す値を返す。アプリケーションに必要なエンコーディング名が MBSTRING バッファ内に指定されている名前と異なる場合は、tpsetmbenc()を使用する。

tuxsetmbaconv()
tuxgetmbaconv()
クライアントプロセスまたはサーバプロセスで、TPMBACONV 環境変数を設定したり取得できるようにする。TPMBACONV が設定されていることを示す値が取得関数から返される場合には、バッファ・タイプスイッチ関数によってコードセットデータ変換が自動的に実行される。tuxsetmbaconv()関数を実行すると、TPMBACONV 環境変数が設定または設定解除される。

tuxsetmbenc()
tuxgetmbenc()
クライアントプロセスまたはサーバプロセスで、TPMBENC 環境変数を設定したり取得できるようにする。アプリケーションでは、設定関数を使用して TPMBENC を設定またはリセットできる。取得関数は、環境リスト内で「TPMBENC=<値>」の形式の文字列を探し、その文字列が存在する場合には、現在の環境での値に対するポインタを返す。
（表 1 の終わり）

表 2: FLD_MBSTRING 関連の関数

（表 2 の始まり）
Fmbpack32()
FML32 API 関数への入力として使用可能なバイトストリームを作成する。コードセットエンコーディング名、コードセットマルチバイトデータ、および入力データ長を入力として受け取る。これらの入力を FML32 で使用可能な形式に変換して格納した出力データへのポインタを返す。

Fmbunpack32()

FLD_MBSTRING に対して FML32 API 関数を実行して得られる出力を受け取り、それをアプリケーションで使用可能な情報に変換する。FML32 関数の実行結果として得られるパック済みのバイトストリームとそのバイト数を、入力として受け取る。コードセットエンコーディング名、マルチバイトユーザデータ、および返されるデータの長さを返す。

tpconvfmb32()
アプリケーション開発者が型付きバッファスイッチ関数とは別個にマルチバイトデータ変換を実行できるようにする。入力用 FML32 バッファ、出力用 FML32 バッファ、および変換先のコードセットエンコーディング名を入力として受け取る。入力用 FML32 バッファをスキャンし、変換先エンコーディング名引数と異なるコードセットエンコーディング名が格納された FLD_MBSTRING フィールド型を更新する。
（表 2 の終わり）

付録 2: サンプル ソフトウェア

マルチバイトデータ変換の例

この例では、簡単な変換シナリオでの MBSTRING 関連 API 関数の使い方を示します。そのサンプルアプリケーションを以下に掲載します。

クライアントサイドアプリケーション

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

#include <stdio.h>
#include "Uunix.h"
#include "atmi.h" /* TUXEDO ヘッダファイル */

#if defined(__STDC__) || defined(__cplusplus)
main(int argc, char *argv[])
#else
main(argc, argv)
int argc;
char *argv[];
#endif
{
/* ************************************************************
このサンプルでは、TOUPPERMB サービスに入力文字列を送信し、その
サービスでそれらの文字を大文字に変換したあと、その結果をこの
クライアントに返す。このクライアントサイドプロセスのエンコーディング名は
UTF-16LE と定義され、送信されるバッファは UTF-8 エンコーディングに
```

```
再定義される。また、サーバサイドのエンコーディングは UTF-16BE になる。
両サイドで自動変換が有効になっていれば、サーバプロセスは MBSTRING を
UTF-8 から UTF-16BE に変換してから TOUPPERMB サービスに渡す。サービスの
処理が終了し MBSTRING が返されると、クライアントプロセスでは、その
MBSTRING が UTF-16BE から UTF-16LE に変換されたあと (このプロセスに定義
されているエンコーディングが UTF-16LE であるため)、その結果を格納した
バッファが tpcall の rcvbuf 引数としてこのアプリケーションに送られる。
最後に、rcvbuf が UTF-8 エンコーディングに再び変換され出力される。
UTF-16LE に変換するステップは必要ないが (すなわち、UTF-8 と UTF-16BE 間
の変換は自動変換で行える)、いくつかの API の使い方を示すために追加
されている。
************************************************************
*/
char *sendbuf, *rcvbuf;
long sendlen, alloclen, rcvlen;
int ret,iolen;

if(argc != 2) {
(void) fprintf(stderr, "Usage: simpclmb string\n");
exit(1);
}

/* クライアントプロセスとして機能する System/T にアタッチする */
if (tpinit((TPINIT *) NULL) == -1) {
(void) fprintf(stderr, "Tpinit failed\n");
exit(1);
}

/*
************************************************************
自動マルチバイト変換を「オフ」にしたほうがよければ、以下の 6 行を
コメントアウトするか削除する。
tuxsetmbaconv は、このクライアントプロセスを制御するだけである。
サーバプロセスでは、独自の環境変数を設定するか、独自の
tuxsetmbaconv()関数を実行する必要がある。
注意: これらの 2 行を使う代わりに、TPMBACONV 環境変数を設定する
方法もある (たとえば、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");
```

```
/*
************************************************************
注意: 以下の 6 行を使う代わりに、TPMBENC 環境変数を設定する
方法もある (たとえば、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]);
/*
************************************************************
注意: このサンプルでは、ASCII 入力文字を使用している。顧客固有の
エンコーディングを使用すると、文字定義における埋め込み NULL が原因で、
OS の提供する文字列関数でうまく処理できないおそれがある。
一般に、メモリ処理またはワイド文字列処理関数は、コードセット
エンコーディングに埋め込み NULL が含まれているかどうかを気にせずに
使用することができる。
したがって、この例では、sendlen に NULL 終端文字用の 1 を加えずに、
bytecnt のままを使用する。文字列関数を使用して送信バッファに
NULL 終端文字を追加するかどうかは、開発者の裁量に任す。
************************************************************
*/
(void) fprintf(stderr,"Input: %s, Length: %d\n", argv[1], sendlen);

/* リクエストと応答用の MBSTRING バッファを確保する */
alloclen = sendlen * 4; /* 最大サイズのバッファを確保しておけば、iconv の最低限の反復実
行に対応できる */
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);

}
/*
************************************************************
新たに tpalloc された sendbuf のデフォルトエンコーディングは UTF-16LE である
 (上記で tuxsetmbenc()を実行したため) が、このクライアントに入力される
データは UTF-8 エンコーディングである (すなわち、argv[1]が UTF-8)。
したがって、ここでは sendbuf のエンコーディングを UTF-8 にリセットする必要がある。
************************************************************
*/

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);

/* TOUPPERMB サービスにリクエストを出し、応答を待つ */
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);

/*
************************************************************
このプロセスが TOUPPERMB サービスから応答バッファを受信した時点で、
rcvbuf は UTF-16BE から UTF-16LE に自動的に変換されている (最初に
tuxsetmbaconv()を呼び出したため) が、このアプリケーションでは
UTF-8 エンコーディングを使ってバッファの内容を出力する必要が
あるため、もう一度 UTF-16LE から UTF-8 に強制的に変換する。
************************************************************
*/
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);
}
/*
************************************************************
注意: tpconvmb では出力用に rcvbuf を再利用し、UTF-8 に変換された
iolen 分のバイトデータを返す。それを文字列として正しく出力する
には、NULL 終端文字を rcvbuf に追加するか、別の char*を使って
そこに strncpy する必要がある。
************************************************************
*/
*(rcvbuf + iolen) = '\0';



/* TOUPPERMB サービスから受信したバッファを UTF-8 に変換したものを出力する */
(void) fprintf(stdout, "simpclmb output string is: %s, Length %d\n",
rcvbuf, iolen);

/* バッファを解放し、System/T から切り離す */
tpfree(sendbuf);
tpfree(rcvbuf);
tpterm();
return(0);
}
```

サーバサイドアプリケーション

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

#include <stdio.h>
#include <ctype.h>
#include <atmi.h>         /* TUXEDO ヘッダファイル */
#include <userlog.h>      /* TUXEDO ヘッダファイル */

/* サーバが起動されると、リクエストの処理を開始する前に
tpsvrinit が実行される。この関数は必ずしも用意する必要はない。
また、tpsvrdone も利用できる (この例では使用されない) が、
```

```
これはサーバの停止時に呼び出される。
*/

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

/* userlog は、一元管理されている TUXEDO メッセージログに書き込む */
userlog("Welcome to the simpservmb server");

/* コンパイラの中には、argc と argv が使用されないと警告を発するものがある */
argc = argc;
argv = argv;
/*
************************************************************
自動マルチバイト変換を「オフ」にしたほうがよければ、以下の 6 行を
コメントアウトするか削除する。
tuxsetmbaconv は、このクライアントプロセスを制御するだけである。
サーバプロセスでは、独自の環境変数を設定するか、独自の
tuxsetmbaconv()関数を実行する必要がある。
注意: これらの 2 行を使う代わりに、TPMBACONV 環境変数を設定する
方法もある (たとえば、export TPMBACONV="YES")。
************************************************************
*/
ret = tuxsetmbaconv(MBAUTOCONVERSION_ON,0);
if(ret == -1) {
        (void) fprintf(stderr, "tuxsetmbaconv failed\n");
        exit(1);
}
        userlog("tuxsetmbaconv ON done");
/*
************************************************************
注意: 以下の 6 行を使う代わりに、TPMBENC 環境変数を設定する
方法もある (たとえば、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);
}

/* この関数は、クライアントから要求されたサービスを実際に実行する。
データバッファへのポインタとそのデータバッファの長さなどを格納した
構造体を引数に取る。
*/
#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);
/*
************************************************************
自動変換はオンになっており (上記の tpsvrinit を参照)、バッファは、
サーバプロセスからこのサービスに送られる前に、サーバプロセスで
定義済みのエンコーディングにすでに変換されている。rqst 内のデータは
現在、UTF-16BE エンコーディングになっているはずで、rqst データの
長さは UTF-8 エンコーディング時の 2 倍になっているだろう。
************************************************************
*/
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");
```

```
/* ***********************************************************
自動マルチバイト変換を「オフ」にし、このアプリケーションで
オンデマンド変換を行うほうがよければ、たとえば、以下の 12 行の
コードを使用するとよい。自動変換が行われず tpconvmb が実行され
ない場合、rqst のデータバイトのエンコーディングは、クライアント
プロセスの場合と同じ定義 (すなわち UTF-8) のままである。
***********************************************************

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 {
/*
***********************************************************
注意: 受信したデータに対して文字列処理/出力関数を好き勝手に使う
のは危険である。すなわち、元のデータは UTF-8 で送信されているが、
受信された変換済みデータはもはや UTF-16BE であり、埋め込み NULL も
含まれている。LIBC の文字列処理/出力関数は正常に動作しないか、
クラッシュするだろう。
***********************************************************
*/
                userlog("TOUPPERMB skip index: %d",i);
        }
}

/* 変換済みのバッファを要求元に返す */
tpreturn(TPSUCCESS, 0, rqst->data, rqst->len, 0);
}
```

FLD_MBSTRING の使い方

最後に、FLD_MBSTRIN の使い方を示す例を紹介します。

クライアントサイドアプリケーション

```
#include <stdio.h>
#include <stdlib.h>
#include <atmi.h>
#include <userlog.h>
#include <fml.h>
#include <fml32.h>
#include "fmltbl32.h"
/*
*****************************************************
fmltbl32.h の生成に使用される fmltbl32 ヘッダファイルには、
以下のようにフィールド定義が 1 つあるだけ。
# 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;
/*
*********************************************************
この例では、同じフィールドが 2 回出現するように UTF-8 形式の
データに設定したあと、そのデータを FML32SRV サービスに送信する。
このサービスは、そのフィールドの入ったバッファを UTF-16BE 形式で
返し、それが元の UTF-8 にローカルに変換される。
*********************************************************
*/
```

```
/* クライアントプロセスとして機能する System/T にアタッチする */
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");

/*
**********************************************************
自動変換がオンになっているので、プロセス環境のエンコーディングを
設定する必要がある。これは、tpcall への応答が元の UTF-8 に
変換されてから、このアプリケーションコードで利用できるように
するためである。
**********************************************************
*/
ret = tuxsetmbenc("UTF-8",0);
if(ret == -1) {
        (void) fprintf(stderr, "tuxsetmbenc failed\n");
        exit(1);
}
(void) fprintf(stderr, "tuxsetmbenc UTF-8 done.\n");

/* FML32 バッファの割り当て */
if ( (fmlptr = (FBFR32 *) tpalloc("FML32", NULL, BUFLEN)) == NULL ) {
        (void) fprintf(stderr,"tpalloc failed: %s\n", tpstrerror(tperrno));
        tpterm();
        exit(1);
}

/* FLD_MBSTRING フィールドに入力されるデータストリームを作成しパックする */
packedlen = 256; /* これは多すぎる。実際に使用されるバイト量は非常に少ない */
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);
}
/* 最初に出現する FLD_MBSTRING フィールド FLD4 を設定する */
```

```
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);
}
/* 2 番目に出現する FLD_MBSTRING フィールド FLD4 を設定する */
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);

/*
**********************************************************
注意: すべてのフィールドは同じエンコーディングを使って定義
されるので、各エンコーディングを別々に設定するのではなく、
FML32 バッファに対して tpsetmbenc(UTF-8)を適用したあと、
flag 引数に FBUFENC、encoding 引数に NULL を指定して Fmbpack32()を
呼び出すこともできよう。そうすれば、バッファの総使用量が少なく
なる。
**********************************************************
*/

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



Fprint32(fmlptr);
userlog("Fchg32 : successful");

/* FML32 バッファを FMLSRV32 サービスに送信する */
if ( tpcall("FMLSRV32", (char *)fmlptr, 0, (char **)&fmlptr, &rlen, 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);
}
```

サーバサイドアプリケーション

```
#include <stdio.h>
#include <ctype.h>
#include <atmi.h>      /* TUXEDO ヘッダファイル */
#include <userlog.h>   /* TUXEDO ヘッダファイル */
#include <fml.h>
#include <fml32.h>
#include "fmltbl32.h"

/* サーバが起動されると、リクエストの処理を開始する前に
tpsvrinit が実行される。この関数は必ずしも用意する必要はない。
また、tpsvrdone も利用できる (この例では使用されない) が、
これはサーバの停止時に呼び出される。
*/

#if defined(__STDC__) || defined(__cplusplus)
tpsvrinit(int argc, char *argv[])
#else
tpsvrinit(argc, argv)
int argc;
char **argv;
#endif
{
int ret=0;
/* コンパイラの中には、argc と argv が使用されないと警告を発するものがある */
argc = argc;
argv = argv;

/* userlog は、一元管理されている TUXEDO メッセージログに書き込む */
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);
}

/* この関数は、クライアントから要求されたサービスを実際に実行する。
データバッファへのポインタとそのデータバッファの長さなどを格納した
構造体を引数に取る。
*/

#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");
/*
*************************************************************
自動変換がオンになっているので、この FMLSRV32 サービスが受信
する FML32 バッファは、すでにローカルエンコーディング (すなわち、
UTF-16BE) に変換されている。以下のコードでは、FML32 バッファ
```

```
からフィールドを取得し、それらのフィールドからユーザデータを
取り出し、そのデータを操作して (すなわち、大文字に変換する)
パックし直し、フィールドを変更したあと、FML32 バッファを
クライアントに送り返す。
*********************************************************
*/

for (occ = 0;occ < 2; occ++) {
buflen = 1024;
        /* FML32 バッファから FLD_MBSTRING フィールドを取得する */
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;
        /* フィールドをアンパックして、ユーザデータとエンコーディング情報に展開する */
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);
        /* 該当するバイトデータを大文字に変換する */



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;
        /* エンコーディング名とユーザデータをパックして、フィールドデータにまとめる */
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);
}
        /* パックされた新しいデータを FLD_MBSTRING フィールドに設定する */
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");

/* 変換済みのバッファを要求元に返す */
tpreturn(TPSUCCESS, 0, rqst->data, 0L, 0);
}
```

カスタム変換関数

```
/*
* CUSTmbconv
*
* この関数は、文字を元のエンコーディング (TCM に定義) から
* 目的のエンコーディングに変換する。
*
* 入力
*       *iptr - 入力バッファへのポインタ
*       ilen - 入力バッファのデータ長
*       *target_enc - iptr 内の文字が、この名前で定義されたエンコーディングに
*                     変換される。
*       *flags - 取り得る値は TMUSEIPTR か TMUSEOPTR。結果の格納先を示す。
*
* 出力
*       *optr - 出力バッファへのポインタ。null の場合は、iptr を出力に使用。
*       olen - 出力バッファのデータ長。optr が null の場合は、ilen を使用。
*       *flags - 取り得る値は TMUSEIPTR か TMUSEOPTR。結果の格納先を示す。
*
* 戻り値
*       -1 - 失敗 (理由についてはエラー番号をチェック)
*       正の整数 - 成功。戻り値は、結果に使用されたバイト数。
*       負の整数 - 領域が不十分。値は、-1 * (必要なバイト数の推測値)。
*/
/*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') ) {
        /* 目的のエンコーディングを指定する引数が見つからない */;
        return(-1); /* 戻り値-1 の tperrno を設定する必要がある */
}

if(tpgetmbenc(iptr,src_enc,0) < 0) {
        /* 元のエンコーディング名が見つからない */;
        return(-1);
}

/* 文字を元のエンコーディング形式から目的のエンコーディング形式に変換する */
cd = iconv_open((const char *)target_enc, (const char *)src_enc);
if (cd == (iconv_t)-1) {
        /* iconv_open が失敗 */
        return (-1);
}

if(optr == NULL) {
/* 出力バッファが指定されず、バッファサイズが不十分なため変換に失敗した */
/* 場合には、再度変換を試みるために送り返されても、入力バッファは使用 */
/* できないであろう。そのため、変換が成功するまでは、出力用の一時バッファを */
/* 使用し、変換に成功したときにそれを入力バッファにコピーする。*/
if(olen == 0) {
        /* おそらく E2BIG エラーが送出され、使用すべき正しいサイズが通知される */
        olen = ilen;
```

```
}
/* olen は、0 でない場合、iptr バッファの最大サイズでなければならない。*/
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); /* 初期状態に戻る */
for ( ;; ) {
ret = iconv(cd, &fptr, &ileft, &tptr, &oleft);
if (ret != (size_t)-1) {
        /* iconv が成功。注意: 変換が必要なかった文字が一部存在する場合もあり、*/



        /* その場合は、入力値の表現と同じになる。*/
        used = used + (olen - oleft);
        olen = oleft;
        if(ileft != 0) {
                /* iconv が実行されていない。再度実行する。*/
        continue;
        }
        if(optr == NULL) {
                /* 出力バッファが指定されていない。tptr バッファを入力バッファにコピーする。
*/
                (void) memcpy(iptr,to,used); /* 危険: iptr のデータ長は used 以上でなけれ
ばならない。*/
                free(to);
                *flags |= TMUSEIPTR;
        } else {
                /* iconv 実行後の文字は出力バッファ optr に格納 */
                *flags |= TMUSEOPTR;
        }
        (void) iconv(cd,NULL,NULL,NULL,NULL); /* 初期状態にリセット */
        (void) iconv_close(cd);
        return (used); /* 出力バッファに使用されたバイト数を返す */
} else {
        /* iconv が失敗 */
        (void) iconv(cd,NULL,NULL,NULL,NULL);/* 初期状態にリセット */
        (void) iconv_close(cd);
```

```
        if(optr == NULL) {
                free(to);
        }
        if (errno == E2BIG) {
                olen = (ilen + (4 * ileft)) * -1;
                *flags |= TMUSEIPTR;
                return (olen); /* 妥当な iptr サイズの推測値を返す */
        } else if (errno == EINVAL) {
                /* 文字/シフトのシーケンスが不完全 */
        } else if (errno == EILSEQ) {
                /* 注意: コードセットの状態に依存するシーケンスは処理しない */
        } else if (errno == EBADF) {
                /* これは、実際には冒頭の iconv_open 時に発生するはず。*/
        } else {
                /* 未定義のエラー */
        }
        return(-1);
}
}
}
```

BEA について

BEA Systems, Inc.（Nasdaq 略称：BEAS）は、「Fortune Global 500」の大半を占める優良企業を始め、世界中で 13,000 社以上の企業ユーザにエンタープライズ・ソフトウェア基盤を提供している、世界最大手のアプリケーションインフラストラクチャソフトウェア企業です。BEA と同社の Tuxedo ならびに WebLogic ブランドは、ビジネスにおける最も信頼性の高い名称の 1 つです。BEA はカリフォルア州サンノゼを本拠とし、世界 33 か国に 91 のオフィスを構えています。Web サイトのアドレスは www.bea.com です。