BEA Tuxedo®

FML を使用した BEA Tuxedo アプリケーションのプログラミング

BEA Tuxedo 8.1

マニュアルの日付 : 2003 **年** 1 **月**

Copyright

Copyright © 2003 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

目次

3. FML および VIEWS の環境設定

4. フィールドの定義と使用

5. フィールド操作関数

6. FML および VIEWS の例

A. FML エラー・メッセージ

このマニュアルについて

このマニュアルでは、BEA Tuxedo ATMI 環境でフィールド操作言語 (FML) 関数を使用する方法について説明します。FML は、フィールド化バッファと呼ばれる記録構造を定義、操作する一連の C 言語関数で、フィールド化バッファにはフィールドと呼ばれる属性値の対が含まれます。

このマニュアルでは、以下の内容について説明します。

- 第 1 章「FML プログラミング入門」では、FML プログラミングの概要を示します。
- 第 2 章「FML および VIEWS の機能」では、FML および VIEWS の機能を説明します。VIEWS を使用すると、フィールド化バッファを C 構造体または COBOL レコードにマッピングできます。
- 第 3 章「FML および VIEWS の環境設定」では、FML および VIEWS の環境設定手順を説明します。
- 第 4 章「フィールドの定義と使用」では、フィールドの定義方法およびフィールドを C 構造体または COBOL レコードにマッピングする方法を示します。
- 第 5 章「フィールド操作関数」では、個々のフィールド操作関数の使用方法を説明します。
- 第 6 章「FML および VIEWS の例」では、FML および VIEWS の例を示します。
- 付録 A 「FML エラー・メッセージ」では、エラー・コードおよびエラー・メッセージのリストを示します。

対象読者

このマニュアルは、ATMI アプリケーションにおける FML 関数の使用方法を学ぶ必要のあるプログラマを対象としています。FML を使用すると、BEA Tuxedo データ・エントリ・プログラムやフィールド化されたデータのプロセス間通信を必要とするプログラムを開発することができます。このマニュアルは、FML を使用したアプリケーションのユーザが、正しく環境を設定するための情報も提供します。

このマニュアルを十分に活用するには、以下の事項について精通している必要があります。

- UNIX システム環境―たとえば、シェル・コマンドや環境変数とは何か、または UNIX ファイルやバックグラウンドでのプロセスの実行とは何かについての十分な知識が必要です。
- C 言語または COBOL 言語―FML を構成する関数およびマクロは、C 言語のプログラムに組み込む要素として設計されています。したがって、C プログラムの開発経験が必要となります。ただし、COBOL (COBOL レコード) で VIEWS を使用する場合、C 言語の知識はほとんど必要ありません。
- BEA Tuxedo について―BEA Tuxedo システムのアプリケーションを使用したことがなくても、少なくとも BEA Tuxedo システムの設計目的を理解し、『C 言語を使用した BEA Tuxedo アプリケーションのプログラミング』または『COBOL を使用した BEA Tuxedo アプリケーションのプログラミング』のアプリケーション開発手順に関する説明を読んでおく必要があります。

e-docs Web サイト

BEA 製品のマニュアルは BEA 社の Web サイト上で参照することができます。BEA ホーム・ページの [製品のドキュメント] をクリックするか、または http://edocs.beasys.co.jp/e-docs/index.html に直接アクセスしてください。

マニュアルの印刷方法

このマニュアルは、ご使用の Web ブラウザで一度に 1 ファイルずつ印刷できます。Web ブラウザの [ファイル] メニューにある [印刷] オプションを使用してください。

このマニュアルの PDF 版は、e-docs Web サイトの BEA Tuxedo マニュアル・ページから入手できます。また、マニュアルの CD-ROM にも収められています。この PDF を Adobe Acrobat Reader で開くと、マニュアル全体または一部をブック形式で印刷できます。PDF 形式を利用するには、BEA Tuxedo Documents ページの [PDF 版] ボタンをクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader をお持ちではない場合は、Adobe Web サイト (http://www.adobe.co.jp/) から無償でダウンロードできます。

関連情報

以下の BEA Tuxedo マニュアルには、FML の使用方法および BEA Tuxedo 環境で FML を使用してアプリケーションを実装する方法についての関連情報が掲載されています。

- 『BEA Tuxedo FML リファレンス』

- 以下の項目については、『BEA Tuxedo のファイル形式とデータ記述方法』の各エントリを参照してください。
  - `compilation(5)` ―アプリケーション・プログラムのコンパイル手順について
  - `field_tables(5)` ― FML フィールド・テーブルについて
  - `viewfile(5)` ― VIEW 記述ファイルの構造について

BEA Tuxedo ATMI およびトランザクション処理の詳細については、「Bibliography」を参照してください。

サポート情報

皆様の BEA Tuxedo マニュアルに対するフィードバックをお待ちしています。ご意見やご質問がありましたら、電子メールで `docsupport-jp@bea.com` までお送りください。お寄せいただきましたご意見は、BEA Tuxedo マニュアルの作成および改訂を担当する BEA 社のスタッフが直接検討いたします。

電子メール メッセージには、BEA Tuxedo 8.1 リリースのマニュアルを使用していることを明記してください。

BEA Tuxedo に関するご質問、または BEA Tuxedo のインストールや使用に際して問題が発生した場合は、`http://www.bea.com` の BEA WebSUPPORT を通して BEA カスタマ・サポートにお問い合わせください。カスタマ・サポートへの問い合わせ方法は、製品パッケージに同梱されている カスタマ・サポート・カードにも記載されています。

カスタマ・サポートへお問い合わせの際には、以下の情報をご用意ください。

- お客様のお名前、電子メール・アドレス、電話番号、Fax 番号
- お客様の会社名と会社の住所
- ご使用のマシンの機種と認証コード
- ご使用の製品名とバージョン

- 問題の説明と関連するエラー・メッセージの内容

表記上の規則

このマニュアルでは、以下の表記規則が使用されています。

規則	項目
太字	用語集に定義されている用語を示します。
Ctrl + Tab	2 つ以上のキーを同時に押す操作を示します。
イタリック体	強調またはマニュアルのタイトルを示します。

等幅テキスト	コード・サンプル、コマンドとオプション、データ構造とメンバ、データ型、ディレクトリ、およびファイル名と拡張子を示します。また、キーボードから入力する文字も示します。
	例 :
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

等幅太字	コード内の重要な単語を示します。 例 : `void **commit** ( )`
等幅イタリック体	コード内の変数を示します。 例 : `String *expr*`

規則	項目
大文字	デバイス名、環境変数、および論理演算子を示します。 例 : LPT1 SIGNON OR
{ }	構文の行で選択肢を示します。かっこは入力しません。
[ ]	構文の行で省略可能な項目を示します。かっこは入力しません。 例 : `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
|	構文の行で、相互に排他的な選択肢を分離します。記号は入力しません。
...	コマンド行で次のいずれかを意味します。 ■ コマンド行で同じ引数を繰り返し指定できること ■ 省略可能な引数が文で省略されていること ■ 追加のパラメータ、値、その他の情報を入力できること 省略符号は入力しません。 例 : `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	コード例または構文の行で、項目が省略されていることを示します。省略符号は入力しません。

1 FML プログラミング入門

ここでは、次の内容について説明します。

- FML とは
- FML を BEA Tuxedo システムで使用する方法
- BEA Tuxedo の型付きバッファ
- FML の用語

FML とは

フィールド操作言語 (FML: Field Manipulation Language) とは、属性と値の組み合わせをフィールドに格納する記憶構造 (フィールド化バッファ) を定義および操作する C 言語関数のセットです。属性はフィールドの識別子であり、対応する値はフィールドのデータ内容を表します。

フィールド化バッファを使用すると、関連したフィールドの集まりに名前でアクセスでき、協調動作するプロセス間でパラメータ化したデータをやり取りする際に非常に便利です。ほかのプロセスと通信する必要のあるプログラムで FML を使用すると、フィールドを格納する構造体を操作せずにフィールドにアクセスすることができます。

FML は、VIEWS と呼ばれる機能も提供します。この機能を使用すると、フィールド化バッファを C 構造体や COBOL レコードにマッピングしたり、逆に C 構造体や COBOL レコードをフィールド化バッファにマッピングできます。VIEWS を使用すると、フィールド化バッファの代わりに構造体内で大量のデータを操作できます。操作対象のデータが構造体に転送されると、アプリケーションの動作が速くなります。つまり、この VIEWS 機能を使用すると、フィールド化バッファのデータ独立性と従来のレコード構造の効率性や簡便性を両方とも実現できます。

FML および VIEWS の機能には、次の 2 つのインターフェイスがあります。

- FML および VIEWS は、16 ビットのフィールド識別子、フィールド長、フィールド・オカレンス、およびレコード長に対応しています。
- FML32 および VIEWS32 は、32 ビットのフィールド識別子、フィールド長、フィールド・オカレンス、およびレコード長に対応しています。このインターフェイスの型定義、ヘッダ・ファイル、関数名、およびコマンド名には、接尾辞として「32」が付きます。

FML を BEA Tuxedo システムで使用する方法

BEA Tuxedo システムでは、ATMI アプリケーションのコンテキストで FML 関数を使用してフィールド化バッファを操作します。

FML 関数は、BEA Tuxedo システムのコア部分を扱うデータ・エントリ・プログラムで使用されます。つまり、これらのプログラムでは、フィールド化バッファを使用して、ユーザが端末から入力したデータをほかのプロセスに転送します。したがって、データ・エントリ・プログラムからフィールド化バッファに入力されたデータを受け取るプログラムを作成する場合も、FML 関数を使用する必要があります。

ユーザの入出力を操作するアプリケーション・プログラムを独自に開発する場合や、プロセス間でメッセージを送受信するプログラムを作成する場合も、プログラム間で送受信されるフィールド化バッファを操作する方法として FML を使用できます。

BEA Tuxedo の型付きバッファ

型付きバッファは、フィールド化バッファという FML の概念に基づいた BEA Tuxedo システムの機能の 1 つです。BEA Tuxedo システムには、FML と VIEW という 2 種類の標準的な型付きバッファが用意されています。これらのバッファのうち、BEA Tuxedo の VIEW バッファの場合は、FML フィールド化バッファとまったく関連付けなくてもよいという点が唯一異なります。

このマニュアルでは、構造化された FML レコードとして VIEW を説明します。ほかのマニュアル、たとえば『C 言語を使用した BEA Tuxedo アプリケーションのプログラミング』では、BEA Tuxedo で提供されるいくつかのバッファ型のうちの 1 つとして VIEW を扱います。

FML の用語

フィールド識別子
: フィールド識別子 (`fldid`) は、FML レコードまたはフィールド化バッファ内の個々のデータ項目を示すタグです。フィールド識別子は、フィールド名 (番号) とフィールドのデータ型で構成されています。

フィールド化バッファ
: フィールド化バッファはデータ構造体であり、構造体内の各データ項目には、データ型とフィールド番号で構成される識別用タグ (フィールド識別子) が対応付けられています。

フィールド型

FML フィールドおよびフィールド化バッファには型が付きます。フィールドの型は、標準 C 言語の型のいずれでもかまいません (short、long、float、double、char)。ほかに、string 型 (NULL 文字で終了する文字列)、carray 型 (文字配列)、mbstring 型 (マルチバイト文字配列 — リリース 8.1 またはそれ以降で使用可能)、ptr 型 (バッファを指すポインタ)、FML32 型 (埋め込み型の FML32 バッファ)、および VIEW32 型 (埋め込み型の VIEW32 バッファ) がサポートされています。mbstring、ptr、FML32、および VIEW32 型は、FML32 インターフェイスでのみサポートされています。COBOL では、COMP-5 型、COMP-1 型、COMP-2 型、および PIC X 型に対応します。ただし、現時点では、COBOL には mbstring、ptr、FML32、および VIEW32 に対応する型はありません。VIEWS では、COBOL COMP-3 との統合のために C 言語のパック 10 進数型もサポートされています。

VIEWS

VIEWS は、フィールド操作言語の機能の一部です。C 構造体のメンバまたは COBOL レコードにフィールドをマッピングすることにより、フィールド化バッファとレコード (C 構造体または COBOL レコード) 間でデータを送受信することができます。フィールド化バッファを大量に操作する場合は、データを C 構造体内に転送するとパフォーマンスが向上します。VIEWS 関数を使用して、フィールド化バッファ内の情報をバッファ内のフィールドから抽出して C 構造体に置き、C 構造体内でデータを操作した後、再度 VIEWS 関数を使用して、更新した値をフィールド化バッファに戻すことができます。FML とは別に VIEWS を単独で使用し、特に COBOL レコードをサポートするためにも使用できます。

2 FML および VIEWS の機能

ここでは、次の内容について説明します。

- レコードをフィールドに分割する : データ構造体とフィールド化バッファ
- FML でのフィールド化バッファの実装方法
- FML の機能
- VIEWS の機能
- FML 関数のエラー処理

レコードをフィールドに分割する : データ構造体とフィールド化バッファ

データ・レコードは、分割できない完全なエンティティでない限り、フィールドに分割できる状態でなければなりません。データ・レコードを分割できないと、レコード内の情報を使用したり、変更できません。ATMI 環境では、次のどちらかを使用してレコードを分割できます。

- C 言語のデータ構造体または COBOL レコード
- フィールド化バッファ

構造体を使用してレコードをフィールドに分割する

レコードを分割する一般的な方法の 1 つは、構造体を使用して連続した記憶域をフィールドに分割することです。フィールドには ID が指定されます。各フィールド内のデータの種類は、データ型の宣言によって示されます。

たとえば、C 言語プログラム内のデータ項目が、従業員の ID、名前、住所、および性別の場合は、次の構造体を使用してプログラムを設定できます。

```
struct S {
       long empid;
       char name[20];
       char addr[40];
       char gender;
};
```

ここでは、empid (従業員の ID) というフィールドが long 型の整数で宣言されています。name (名前) は 20 文字の文字配列で宣言され、addr (住所) は 40 文字の文字配列で宣言されます。gender (性別) は、単一文字 (m と f など) で宣言されます。

C プログラム内で、変数 p が struct S 型の構造体を指す場合は、p–>empid、p–>name、p–>addr、および p–>gender という参照を使用してフィールドを指定できます。

同じデータ構造の COBOL COPY ファイルは、以下のようになります (第 01 行はアプリケーション側で提供されます)。

```
05 EMPID                          PIC S9(9) USAGE IS COMP-5.
05 NAME                           PIC X(20).
05 ADDR                           PIC X(40).
05 GENDER                         PIC X(01).
05 FILLER                         PIC X(03).
```

COBOL のプログラムで、第 01 行に MYREC という名前を付けた場合は、EMPID IN MYREC、NAME IN MYREC、ADDR IN MYREC、および GENDER IN MYREC を使用して各フィールドにアクセスできます。

このデータ表現法は広く使用されており、通常は適用できますが、次の 2 つの潜在的な問題があります。

- データ構造体が変更されるたびに、その構造体を使用するすべてのプログラムを再コンパイルする必要があります。

- 構造体のサイズと構成要素のフィールドのオフセットは、すべて固定されています。そのため、(a) 常にすべてのフィールドに値が入るわけではないこと、および、(b) フィールドは最もサイズの大きいエントリに合わせてサイズ調整されやすいこと、が原因で、無駄な領域が生じる場合があります。

フィールド化バッファを使用してレコードをフィールドに分割する

レコードをフィールドに分割する別の方法は、フィールド化バッファを使用することです。

フィールド化バッファは、レコードのフィールドに対して連想アクセスを提供するデータ構造体です。つまり、フィールド名は、フィールドのデータ型のほか、記憶域の位置を含む識別子と関連付けられています。

フィールド化バッファの主な利点は、データの独立性です。つまり、バッファに対してフィールドを追加または削除したり、フィールド長を変更しても、フィールドを参照するプログラムを再コンパイルする必要はありません。データの独立性を実現するため、フィールドは以下のように使用されます。

- フィールドは、レコードの構造体で規定されている固定のオフセットではなく、識別子によって参照されます。
- フィールドへのアクセスは、関数の呼び出しによってのみ可能です。

ATMI 環境では、協調動作するプロセス間で送信されるデータを表現する標準的な方法として、フィールド化バッファを使用できます。

FML でのフィールド化バッファの実装方法

フィールド化バッファの作成、更新、アクセス、入力、出力、および操作は、フィールド操作言語 (FML) によって行います。FML には以下の特徴があります。

- フィールド化バッファの作成および操作のための便利で標準的な規則を提供します。
- フィールド化バッファを使用するプログラムでデータの独立性を実現できます。

FML は、C プログラムから呼び出すことができる関数およびマクロのライブラリとして実装されています。FML には、以下の関数が用意されています。

- フィールド化バッファを作成、更新、アクセス、および操作するための関数
- フィールド化バッファの構造体に対する入力または出力を行うときにデータ型を変換するための関数
- フィールド化バッファと C 構造体、またはフィールド化バッファと COBOL レコードの間でデータを転送するための関数

上記のうち、最後のセットが FML VIEWS ソフトウェアを構成する関数のセットです。VIEWS は、FML フィールド化バッファとアプリケーション・プログラムの構造体 (C 言語または COBOL 言語) との間でデータを交換する関数のセットです。プログラムは、別のプロセスからフィールド化バッファを受け取ると、次の動作として以下のいずれかを実行します。

- FML 関数呼び出しを使用して、バッファ内でバッファのデータを直接操作します (COBOL では使用できません)。
- VIEWS 関数を使用してフィールド化バッファから構造体にデータを転送し、次に、通常の C または COBOL 文によって構造体内のデータを処理します。

バッファのデータに対して、時間のかかる操作を行う必要がある場合は、フィールド化バッファのデータを構造体に転送し、通常の C または COBOL 文を使用してそのデータを処理すると、プログラムのパフォーマンスを向上させることができます。次に、再び VIEWS 関数を使用してデータをフィールド化バッファに戻すと、そのバッファを別のプロセスに送信できます。

VIEWS を使用する前に、受信するフィールド化バッファのデータ形式がプログラム側で認識できるようにプログラムを設定しておかなければなりません。この設定には、システム・キャッシュに保存されている VIEW 記述のセットを使用します。

VIEW 記述は、ソース VIEW ファイル内に作成および格納されます。VIEW 記述により、フィールド化バッファ内のフィールドが C 構造体または COBOL レコード内のメンバにマッピングされます。ソース VIEW 記述がコンパイルされると、その記述は、フィールド化バッファと C 構造体、またはフィールド化バッファと COBOL レコードの間で転送されるデータをマッピングするために使用できます。

主要なファイルに VIEW 記述をキャッシングしておくと、プログラムのデータ独立性を強化できます。つまり、VIEWS 記述を変更し、再コンパイルするだけで、VIEWS を使用するアプリケーション・プログラム全体のデータ形式に変更を加えることができます。

FML の機能

ここでは、次の内容について説明します。

フィールド化バッファとは

フィールド化バッファとは、レコードのフィールドに対して連想アクセスを行うデータ構造です。

FML フィールド化バッファ内の各フィールドには、フィールドのデータ型に関する情報と一意な ID 番号を組み合わせた short 型の整数がラベル付けされています。このラベルは、フィールド識別子 (fldid) と呼ばれます。可変長項目の場合は、fldid の後にデータ長を指定します。

フィールド化バッファは、fldid、データ、の順で組み合わせるか、または、可変長項目の場合は fldid、フィールド値長、データ、の順で組み合わせて表現します。次の図を参照してください。

図 2-1 フィールド化バッファ

FML 関数を使用するたびに #include を使用して組み込まれるヘッダ・ファイル (fml.h または fml32.h) では、typedef により、フィールド識別子は FLDID (FML32 では FLDID32)、フィールド値長は FLDLEN (FML32 では FLDLEN32)、フィールド・オカレンス数は FLDOCC (FML32 では FLDOCC32) と定義されます。

サポートされているフィールド型

サポートされているフィールド型は、short 型、long 型、float 型、double 型、char 型、string 型、carray 型 (文字配列)、mbstring 型 (マルチバイト文字配列 —Tuxedo リリース 8.1 またはそれ以降で使用可能)、ptr 型 (バッファを指すポインタ)、FML32 型 (埋め込み型の FML32 バッファ)、および VIEW32 型 (埋め込み型の VIEW32 バッファ) です。mbstring、ptr、FML32、

および VIEW32 型は、FML32 インターフェイスでのみサポートされています。これらの型は、次のリストに示すように、fml.h (または fml32.h) 内の #define 文で定義されています。

コード リスト 2-1　fml.h **および** fml32.h **における** FML **フィールド型の定義**

```
#define FLD_SHORT       0       /* short int */
#define FLD_LONG        1       /* long int */
#define FLD_CHAR        2       /* character */
#define FLD_FLOAT       3       /* single-precision float */
#define FLD_DOUBLE      4       /* double-precision float */
#define FLD_STRING      5       /* string - null terminated */
#define FLD_CARRAY      6       /* character array */
#define FLD_PTR         9       /* pointer to a buffer */
#define FLD_FML32      10       /* embedded FML32 buffer */
#define FLD_VIEW32     11       /* embedded VIEW32 buffer */
#define FLD_MBSTRING   12       /* multibyte character array */
```

FLD_STRING、FLD_CARRAY、および FLD_MBSTRING はいずれも配列ですが、次の点が異なります。

- FLD_STRING は、NULL 以外の文字で構成し、NULL で終了する可変長の配列です。
- FLD_CARRAY または FLD_MBSTRING は、NULL も NULL 以外の文字も使用できる可変長のバイト配列です。

フィールドを追加または変更する関数には、FLDLEN 引数を指定します。この引数は、FLD_CARRAY または FLD_MBSTRING フィールドを処理する場合は必須です。文字列または文字配列のサイズは、FML では最大 65,535 文字、FML32 では最大 2 ギガバイトに制限されています。マルチバイト文字配列のサイズは、FML32 で最大 ??? ギガバイトに制限されています。

unsigned 型のデータ型は、フィールド化バッファに格納しないでください。このデータ型を使用すると、フィールド化バッファからデータを検索するたびに、FML の変換関数を使用して、unsigned short 型データをすべて long 型データに変換するか、またはそれらのデータを正しい unsigned 型にキャストしなければなりません。

ほとんどの FML 関数では、型のチェックは行われません。通常、フィールド化バッファで更新または検索された値は、ネイティブな型と一致すると見なされます。たとえば、バッファ・フィールドが FLD_LONG として定義されていると、必ず、long 型の値のアドレスを渡す必要があります。FML 変換関数は、データをフィールド化バッファに格納する (または、データをフィールド化バッファから検索する) ほか、ユーザ指定の型からネイティブ・フィールド型 (または、ネイティブ・フィールド型からユーザ指定の型) にデータを変換します。

FLD_PTR フィールド型を使用すると、FML32 バッファ内のアプリケーション・データへのポインタを埋め込むことができます。アプリケーション側では、データ・バッファへのポインタの追加、変更、アクセス、および削除が可能です。FLD_PTR フィールドが指すバッファは、tpalloc(3c) を呼び出して割り当てます。FLD_PTR フィールド型は、FML32 でのみサポートされています。

FLD_FML32 フィールド型は、レコード全体を単一フィールドとして FML32 バッファに格納できます。同様に、FLD_VIEW32 フィールド型は、C 構造体全体を FML32 バッファ内に単一フィールドとして格納できます。FLD_FML32 および FLD_VIEW32 フィールド型は、FML32 でのみサポートされています。

VIEWS の int 型

VIEWS は、大半の FML 関数でサポートされているデータ型のほか、ソース VIEW 記述内で int 型を間接的にサポートします。VIEW 記述がコンパイルされると、VIEW のコンパイラは、マシンの種類に応じて、すべての int 型を short 型か long 型に自動的に変換します。詳細については、2-13 ページの「VIEWS の機能」を参照してください。

VIEWS の dec_t 型

VIEWS では、ソース VIEW 記述内で dec_t パック 10 進数型もサポートされています。このデータ型は、VIEW の構造体から COBOL のプログラムへの転送に便利です。dec_t 型を使用する C プログラムでは、『BEA Tuxedo C リ

ファレンス』の「decimal(3c)」リファレンス・ページで説明されている関数を使用してフィールドの初期化およびアクセスを行う必要があります。COBOL プログラムでは、パック 10 進数 (COMP-3) の定義を使用してこのフィールドに直接アクセスできます。FML では dec_t フィールドがサポートされていないため、このフィールドは、VIEW から FML に変換するときに、フィールド化バッファ内の対応する FML フィールドのデータ型 (string 型など) に自動的に変換されます。

フィールド名から識別子へのマッピング

BEA Tuxedo システムでは、フィールドは通常、フィールド識別子 (fldid) の整数によって参照されます。フィールド識別子の詳細については、4-2 ページの「フィールド名とフィールド識別子を定義する」を参照してください。このため、変更の可能性があるフィールド名を使用しないで、プログラム内でフィールドを参照できます。

識別子をフィールド名に割り当てる (マッピングする) には、次のいずれかの方法を使用します。

- フィールド・テーブル・ファイル (通常の UNIX ファイル)
- C 言語のヘッダ (#include) ファイル

通常のアプリケーション・プログラムでは、上記の方法のどちらか、または両方を使用して、フィールド識別子をフィールド名にマッピングできます。

FML でフィールド化レコード内のデータにアクセスするには、FML でフィールド名と識別子のマッピング情報にアクセスできなければなりません。FML は、以下の方法のいずれかを使用してこのマッピング情報を取得します。

- 実行時に、UNIX フィールド・テーブル・ファイルと FML マッピング関数を使用します。
- コンパイル時に C ヘッダ・ファイルを使用します。

フィールド名と識別子のマッピングは、COBOL では使用できません。

実行時 : フィールド・テーブル・ファイル

フィールド・テーブル・ファイルにより、FML プログラムの実行時、フィールド名と識別子のマッピング情報が使用可能になります。ただし、フィールド名と識別子のマッピング・テーブル・ファイルの場所は、2 つの環境変数を用いて、プログラマが直接設定します。

環境変数 FLDTBLDIR には、フィールド・テーブルの検索用ディレクトリのリストを指定し、FIELDTBLS 環境変数には、それらのテーブル・ディレクトリから検索すべきファイルのリストを指定します。対応する FML32 の環境変数は、FLDTBLDIR32 および FIELDTBLS32 です。

アプリケーション・プログラム内では、FML 関数である Fldid() により、フィールド名からフィールド識別子への変換が実行時に行われます。フィールド識別子をフィールド名に変換する処理は、Fname() によって行われます (Fldid(3fml) および Fname(3fml) を参照してください。FML32 については Fldid32 および Fname32 を参照してください)。これらの 2 つの関数のうち、最初に呼び出された関数によってメモリ内の領域がフィールド・テーブル用に動的に割り当てられ、フィールド・テーブルがプロセスのアドレス領域にロードされます。フィールド・テーブルが不要になると、領域を回復できます。詳細については、4-8 ページの「フィールド・テーブルをロードする」を参照してください。

アプリケーション内でフィールド名と識別子のマッピングの変更が予測される場合は、この方法を使用します。詳細については、4-1 ページの「フィールドの定義と使用」を参照してください。

コンパイル時 : ヘッダ・ファイル

mkfldhdr() (または mkfldhdr32()) を使用すると、フィールド・テーブル・ファイルからヘッダ・ファイルを作成できます。これらのヘッダ・ファイルは、C プログラムの #include に組み込まれており、フィールド名とフィールド識別子のマッピングを行います。mkfldhdr、mkfldhdr32(1) の詳細については、『BEA Tuxedo コマンド・リファレンス』を参照してください。

C プリプロセッサは、コンパイル時にフィールド・ヘッダ・ファイルを使用して、すべてのフィールド名の参照をフィールド識別子に変換します。したがって、前の節で説明したように、フィールド・テーブル・ファイルを使用して `Fldid()` 関数や `Fname()` 関数を呼び出す必要はありません。

プログラムで必要なフィールド名が常にわかる場合は、フィールド・テーブルのヘッダ・ファイルを `#include` でプログラムに組み込むと、データ領域を節約できます。データ領域を節約すると、プログラムではすばやく目的のタスクを実行できます。

ただし、この方法はコンパイル時にマッピングを解決するため、アプリケーション内でフィールド名とフィールド識別子のマッピングに変更が生じる可能性がある場合は使用しないでください。詳細については、4-1 ページの「フィールドの定義と使用」を参照してください。

フィールド化バッファのインデックス

フィールド化バッファに多数のフィールドが含まれる場合、内部インデックスを使用すると、FML でのアクセスが促進されます。通常、ユーザはこのインデックスの存在を意識することはありません。

ただし、フィールド化バッファのインデックスは、メモリおよびディスク領域を消費します。したがって、フィールド化バッファをディスクに格納したり、プロセス間またはコンピュータ間で転送する場合は、まず、このインデックスを削除すると、ディスク領域や転送時間を節約できます。

インデックスを削除するには、`Funindex()` 関数を使用します。フィールド化バッファがディスクから読み取られるか、または、送信プロセスによって受け取られると、`Findex()` 関数を使用して再びインデックスを明示的に作成できます。

ただし、この方法を使用してもメモリは節約できません。`Funindex()` 関数を使用しても、フィールド化バッファのインデックス用に使用されるインコア・メモリは回復できません。

詳細については、『BEA Tuxedo FML リファレンス』の「`Funindex、Funindex32(3fml)`」、または「`Findex、Findex32(3fml)`」を参照してください。

複数のオカレンスを持つフィールド化バッファのフィールド

フィールド化バッファ内のフィールドは、複数回出現する場合があります。FML 関数の多くが、検索または変更の対象にするフィールド・オカレンスを指定する引数を取ります。フィールドが複数回出現する場合、オカレンスは、最初のオカレンスを 0 として、順次番号付けされます。オカレンスのセットは論理的なシーケンスを構成しますが、オカレンス番号に関連するオーバヘッドは発生しません (つまり、オカレンス番号はフィールド化バッファに格納されません)。

フィールドのオカレンスを追加すると、そのオカレンスは、オカレンスのセットの最後に追加され、最も大きい番号の次の番号が指定されたオカレンスとして参照されます。最も大きい番号が指定されたオカレンス以外のオカレンスを削除すると、削除されたオカレンスより大きい番号のオカレンスには、1 つ低い番号が指定されます。たとえば、オカレンス 6 はオカレンス 5 になり、オカレンス 5 はオカレンス 4 になります。

論理式とフィールド化バッファ

アプリケーション・プログラムが次に行うアクションは、ユーザ端末やデータベース・レコードなど、別のソースから受け取ったフィールド化バッファの 1 つまたは複数のフィールドの値によって決まります。FML には、フィールド化バッファや VIEW に関する論理式を作成して、指定したバッファや VIEW が論理式の条件に合致するかどうかを判定するための関数がいくつか用意されています。

作成された論理式は、評価ツリーにコンパイルされます。評価ツリーは、フィールド化バッファまたは VIEW が指定された論理式の条件を満たすかどうかの判定に使用されます。

たとえば、プログラムがフィールド化バッファ (バッファ A) にデータ・レコードを読み込み、そのバッファに論理式を適用するとします。バッファ A が論理式で指定された条件を満たす場合は、FML 関数が使用され、別のバッファ (バッファ B) をバッファ A からのデータで更新します。

VIEWS の機能

VIEWS は、プログラムがフィールド化バッファを受け取った後またはフィールド化バッファを別のプログラムに送る前に、バッファのデータに対して大量の処理を行う場合に特に便利です。

このような場合、VIEWS 関数を使用してフィールド化バッファのデータをバッファから C 構造体に転送し、その後でデータを処理すると、処理効率を高めることができます。フィールドをバッファ内で処理する場合、FML 関数より C 関数の方が処理時間が短いためです。C 構造体でデータの処理を終了したら、そのデータをフィールド化バッファに戻し、さらに別のプログラムに転送できます。

VIEWS には、以下の特徴があります。

- C 構造体とフィールド化バッファのマッピング、または COBOL レコードとフィールド化バッファのマッピングを指定するソース VIEW 記述を作成し、構造体とバッファ間のデータ転送を可能にします。
- VIEW コンパイラである `viewc` (または `viewc32`) を使用すると、実行時にアプリケーション・プログラムによって解釈されるオブジェクト VIEW 記述 (バイナリ形式のファイルに格納) を生成できます。このコンパイラは、C プログラムで使用されると VIEW 記述で使用した構造体を定義できるヘッダ・ファイルも生成します。また、必要に応じて、COBOL プログラムで使用できる COPY ファイルを生成して、VIEW 記述で使用するレコードを定義することもできます。VIEW コンパイラの詳細については、『BEA Tuxedo コマンド・リファレンス』の「viewc、viewc32(1)」を参照してください。

- VIEW 逆アセンブラを使用すると、オブジェクト VIEW 記述を読み取り可能な形式に変換する (つまり、ソース VIEW 記述に戻す) ことができます。逆アセンブラの出力は、VIEW コンパイラに再入力できます。
- C 構造体または COBOL レコードからフィールド化バッファへのデータ転送は、FUPDATE、FJOIN、FOJOIN、FCONCAT、の各モードで実行できます。これらのモードは、Fupdate、Fupdate32(3fml)、Fjoin、Fjoin32(3fml)、Fojoin、Fojoin32(3fml)、および Fconcat、Fconcat32(3fml) の各 FML 関数によってサポートされているモードと似ています。
- オブジェクト VIEW 記述は、実行時に要求に応じて VIEW ファイル・キャッシュに読み込まれ、キャッシュがいっぱいになるまでキャッシュ内に保存されます。キャッシュがいっぱいになり、キャッシュ内にないオブジェクト VIEW 記述が必要になると、アクセスされた時期が最も古いオブジェクト VIEW 記述がキャッシュから削除され、新しいオブジェクト VIEW 記述のための領域となります。
- VIEW 記述では、FML でサポートされているすべての型 (FLD_PTR、FLD_FML32、および FLD_VIEW32 を除く) を使用できます。さらに、整数およびパック 10 進数もサポートされています。
- フィールド化バッファと構造体の間でデータを転送すると、転送元のデータは、自動的に宛先データの型に変換されます。たとえば、文字列フィールドを整数メンバにマップする場合は、Ftypcvt() によって文字列が自動的に整数に変換されます。詳細については、『BEA Tuxedo FML リファレンス』の「Ftypcvt、Ftypcvt32(3fml)」を参照してください。
- 複数のフィールドのオカレンスをサポートしています。
- VIEW 記述では、ユーザ指定およびデフォルトの NULL 値がサポートされています。
- VIEW 内のアプリケーション・データに対する論理式をコンパイルおよび評価する関数が用意されています。

ソース VIEW ファイルは、1 つまたは複数のソース VIEW 記述を含む一般的なテキスト・ファイルです。ソース VIEW ファイルは、VIEW コンパイラである viewc または viewc32 への入力として使用され、ソース VIEW 記述をコ

ンパイルし、オブジェクト VIEW ファイルに格納します。VIEW コンパイラの詳細については、『BEA Tuxedo コマンド・リファレンス』の「viewc、viewc32(1)」を参照してください。

VIEW コンパイラは、オブジェクト VIEW ファイル用の C ヘッダ・ファイルも生成します。これらのヘッダ・ファイルは、アプリケーション・プログラム内に組み込まれると、オブジェクト VIEW 記述で指定された構造体を定義できます。

VIEW コンパイラは、必要に応じてオブジェクト VIEW ファイルに対する COBOL COPY ファイルも生成します。これらの COPY ファイルは、COPY プログラムに組み込み、オブジェクト VIEW 記述で指定するレコード形式を定義できます。

NULL 値は構造体の空のメンバを表すのに使用し、viewfile の構造体の各メンバに対して指定できます。メンバに NULL 値を指定しないと、デフォルトの NULL 値が使用されます。

ただし、構造体からフィールド化バッファへの転送時には、NULL 値を含む構造体メンバは転送されません。

また、C または COBOL 構造体のメンバとフィールド化バッファ内のフィールドの間にマッピングが存在していても、それらの間でのデータ転送を禁止することができます。これは、ソース VIEW ファイルで指定します。

FML VIEWS 関数として、`Fvstof()`、`Fvftos()`、`Fvnull()`、`Fvopt()`、`Fvselinit()`、`Fvsinit()` があります。COBOL では、VIEWS 機能によって `FVSTOF` および `FVFTOS` の 2 つの手順が提供されています。どちらの VIEW 関数を呼び出しても、指定されたオブジェクト VIEW ファイルが検索され、そのファイルが見つかると、VIEW ファイルのキャッシュに自動的にロードされます。つまり、環境変数 `VIEWFILES` で指定された各ファイルが順番に検索され (3-1 ページの「FML および VIEWS の環境設定」参照)、指定された名前が付いた最初のオブジェクト VIEW ファイルがロードされます。同じ名前を持つ以降のオブジェクト VIEW ファイルは無視されます。FML VIEWS 関数の詳細については、『BEA Tuxedo FML リファレンス』を参照してください。

VIEWS では、構造体の配列、ポインタ、共用体、typedef はサポートされていません。

複数のオカレンスを持つ VIEWS のフィールド

VIEWS は、フィールド化バッファと C 構造体、またはフィールド化バッファと COBOL レコードの間で交換されるフィールドを扱うため、バッファ内で複数回発生するフィールドも処理する必要があります。

構造体内にフィールドの複数のオカレンスを格納するため、メンバは、C 言語または COBOL の OCCURS 句により配列として宣言されます。フィールドの 1 オカレンスに対して 1 つの配列の要素が使用されます。配列のサイズは、バッファ内のフィールド・オカレンスの最大数を反映します。

フィールド化バッファから C 構造体または COBOL レコードへのデータ転送時に、受信側の配列に存在する要素の数がフィールド化バッファ内のオカレンスの数より多い場合、余分な要素には、デフォルト値またはユーザ指定の NULL 値が割り当てられます。バッファ内のオカレンスの数が配列内の要素の数より多い場合は、バッファ内の余分なオカレンスは無視されます。

C 構造体または COBOL レコードからフィールド化バッファにデータを転送する場合、デフォルト値またはユーザ指定の NULL 値に等しい値を含む配列メンバは無視されます。

FML 関数のエラー処理

FML 関数は、エラーを検出すると、以下の値のいずれか 1 つを返します。

- ポインタを返す関数の場合は、NULL を返します。
- `FLDID` を返す関数の場合は、`BADFLDID` を返します。
- 上記以外のすべての関数は、`-1` を返します。

FML 関数の呼び出しに対するすべての戻り値はこれらの条件と照合され、その結果、エラーが検出されます。

エラーの場合はすべて、外部整数 Ferror が fml.h で定義されたエラー番号に設定されます。FML32 では、Ferror32 が fml32.h で定義されたエラー番号に設定されます。

F_error() 関数 (または F_error32() 関数) は、標準エラー出力上にメッセージを出力します。この関数は、パラメータを 1 つ (文字列) とり、コロンと空白文字を追加してその引数文字列を出力します。次に、エラー・メッセージとその後に続く復帰改行文字を出力します。表示されるエラー・メッセージは、エラー発生時に設定された Ferror 内の現在のエラー番号に対して定義されているメッセージです。

大抵の場合、F_error() 関数 (または F_error32() 関数) の引数文字列には、エラーを引き起こしたプログラムの名前が含まれます。詳細については、『BEA Tuxedo FML リファレンス』の「F_error、F_error32(3fml)」を参照してください。

エラー・メッセージのテキストをメッセージ・カタログから取り出すため、Fstrerror、Fstrerror32(3fml) を使用できます。これらの関数は、userlog(3c)、F_error()、または F_error32() の引数として使用できるポインタを返します。

FML の関数のエラー・コードについては、『BEA Tuxedo FML リファレンス』の各関数のエントリを参照してください。

3 FML および VIEWS の環境設定

ここでは、次の内容について説明します。

- FML および VIEWS の環境設定の要件
- FML のディレクトリ構造
- FML および VIEWS で使用される環境変数

FML および VIEWS の環境設定の要件

FML フィールド化バッファの操作を開始したり、構造体とフィールド化バッファ間でフィールドを移動する VIEWS 関数を使用するには、まず、必要な環境変数を設定して、これらの方式が環境に組み込まれるように設定します。この節では、その方法について説明します。

FML のディレクトリ構造

BEA Tuxedo システムに同梱される FML ソフトウェアは、ローカル・ファイル・システムのサブツリーに格納されています。いくつかの FML モジュールでは、このサブツリー構造が使用されます。TUXDIR 環境変数には、BEA Tuxedo ATMI サーバがインストールされているディレクトリの絶対パス名を設定しておかなければなりません。

BEA Tuxedo のインストール先ディレクトリは、以下のサブディレクトリで構成されています。

- include ― C アプリケーション・コードの作成者に必要なヘッダ・ファイルが格納されています。
- cobinclude ― COBOL アプリケーション・コードの作成者に必要な COPY ファイルが格納されています。このディレクトリの名前は、ファイル名が 8.3 に制限されているオペレーティング・システムでは cobinclu になります。
- bin ― FML の実行可能コマンドが格納されています。
- lib ― FML のサブルーチン・パッケージが格納されています。FML 関数を使用するプログラムをコンパイルする場合は、$TUXDIR/lib/libfml.*suffix* と $TUXDIR/lib/libgp.*suffix* を C コンパイラのコマンド行で指定して、外部参照を解決する必要があります。libfml32.*suffix* には、FML32 関数および VIEW32 関数が含まれています。接尾辞は、共用オブジェクトを使用しない POSIX オペレーティング・システムでは .a、共用オブジェクトを使用する場合は .so.*release*、Windows では .lib であり、ダイナミック・リンク・ライブラリを使用するプラットフォームの Tuxedo システムの DLL の一部です。

FML を使用する C アプリケーションには、以下のヘッダ・ファイルをここに示す順序でインクルードします。

```
#include <stdio.h>
#include “fml.h”
```

fml.h または fml32.h ファイルには、FML ソフトウェアで使用される構造体、シンボリック定数、およびマクロのための定義が格納されています。

FML および VIEWS で使用される環境変数

FML および VIEWS で使用される環境変数は以下のとおりです。

- FML では、次の変数を使用して、システム生成されたファイルを検索します。
  - TUXDIR ―この変数は、FML を含む BEA Tuxedo システム・ソフトウェアのインストール先ディレクトリの最上位ノードに設定する必要があります。
- FML では、次の変数を使用して、フィールド・テーブル・ファイルにアクセスします。
  - FIELDTBLS ―この変数には、所定のアプリケーション・プログラム用のフィールド・テーブル・ファイルをカンマで区切ってリストします。絶対パス名で指定したファイルは、そのまま使用されます。相対パス名で指定したファイルは、FLDTBLDIR 変数で指定したディレクトリのリストから検索されます。FML32 では、FIELDTBLS32 が使用されます。FIELDTBLS を設定しないと、単一ファイル名 fld.tbl が使用されます。(この場合でも、FLDTBLDIR は適用されます。以下の説明を参照してください。
  - FLDTBLDIR ―この変数には、相対ファイル名を持つフィールド・テーブル・ファイルの検索に使用するディレクトリをコロンで区切ってリストします。使用法は PATH 環境変数と同様です。FLDTBLDIR を設定しない場合、または NULL の場合、カレント・ディレクトリの値が使用されます。FML32 では、FLDTBLDIR32 が使用されます。

  詳細については、4-1 ページの「フィールドの定義と使用」を参照してください。

- VIEWS 機能では、FML と同じ環境変数 (FLDTBLDIR および FIELDTBLS) のほかに、2 つの環境変数が使用されます。
  - VIEWFILES ―この変数には、アプリケーション・プログラムで使用するオブジェクト VIEW ファイルをカンマで区切ってリストします。絶対パス名で指定したファイルは、そのまま使用されます。相対パス名で指定したファイルは、VIEWDIR 変数で指定したディレクトリのリストから検索されます (次の項目を参照)。VIEW32 では、VIEWFILES32 が使用されます。
  - VIEWDIR ―この変数には、相対ファイル名を指定した VIEW オブジェクト・ファイルを検索するために使用する、コロンで区切ったディレクトリのリストを指定します。この変数の設定方法と使用方法は、PATH 環境変数と同じです。VIEWDIR を設定しない場合、または NULL の場合、カレント・ディレクトリの値が使用されます。VIEW32 では、VIEWDIR32 が使用されます。
- FML32 では、次の変数を使用して、フィールド型 FLD_MBSTRING をサポートします。
  - TPMBENC ―この変数には、BEA Tuxedo 8.1 またはそれ以降が動作しているアプリケーション・サーバまたはクライアントで FML32 型付きバッファ内の FLD_MBSTRING フィールド用に設定されているコード・セット符号化名を指定します。アプリケーション・サーバまたはクライアントのプロセスが、FLD_MBSTRING フィールドを含んだ FML32 バッファを割り当ておよび送信する際に、Fmbpack32() の *enc* 引数が未定義で *flag* 引数が FBUFENC に設定されていない場合、TPMBENC で定義したコード・セット符号化名が自動的に使用されます。

    アプリケーション・サーバまたはクライアントのプロセスが、FLD_MBSTRING フィールドを含んだ FML32 バッファを受信する際、別の環境変数 TPMBACONV が設定されていると想定される場合、TPMBENC で定義したコード・セット符号化名は、受信側バッファ内の FLD_MBSTRING フィールドに指定されているコード・セット符号化名と自動的に比較されます。符号化名が異なる場合、FLD_MBSTRING フィールドのデータは、TPMBENC で定義した符号化に自動的に変換されてから、サーバまたはクライアントのプロセスに配信されます。

TPMBENC にはデフォルト値はありません。FLD_MBSTRING フィールドを使用するアプリケーション・サーバまたはクライアントでは、自動変換が機能するように TPMBENC を定義する必要があります。

注記　TPMBENC の使用方法は、MBSTRING 型付きバッファの場合とほぼ同じです。

- TPMBACONV ― この変数には、BEA Tuxedo 8.1 またはそれ以降が動作しているアプリケーション・サーバまたはクライアントが、受信側 FML32 バッファ内の FLD_MBSTRING フィールドを TPMBENC で定義された符号化に自動的に変換するかどうかを指定します。デフォルトでは、自動変換は無効になっています。つまり、FLD_MBSTRING フィールドのデータは、符号化変換が行われずに送信先サーバまたはクライアントのプロセスに配信されます。TPMBACONV を Y (yes) など NULL 以外の値に設定すると、自動変換が有効になります。

  注記　TPMBACONV の使用方法は、MBSTRING 型付きバッファの場合とほぼ同じです。

詳細については、5-66 ページの「FLD_MBSTRING フィールドの変換」を参照してください。

4 フィールドの定義と使用

ここでは、次の内容について説明します。

- FML および VIEWS を使用するための準備
- FML および VIEWS のフィールドの定義
- フィールドを C 構造体および COBOL レコードにマッピングする

FML および VIEWS を使用するための準備

FML フィールド化バッファを操作したり、VIEWS 関数を使用して構造体とフィールド化バッファ間でフィールドを移動するには、次の準備作業が必要です。

- フィールドを定義します。
- フィールド定義をアプリケーション・プログラムで使用できるようにします。実行時にフィールド・テーブル・ファイルおよびマッピング関数を使用するか、またはコンパイル時に C ヘッダ・ファイルを使用します。

- ソース VIEW 記述をオブジェクト VIEW 記述にコンパイルし、対応する C ヘッダ・ファイルおよび COBOL COPY ファイルを生成します。

この章では、これらの内容と、関連する処理について説明します。

FML および VIEWS のフィールドの定義

ここでは、次の内容について説明します。

- フィールド名とフィールド識別子を定義する
- フィールド・テーブル・ファイルを作成する
- フィールド名からフィールド識別子にマッピングする
- フィールド・テーブルをロードする
- フィールド・テーブルからヘッダ・ファイルに変換する

フィールド名とフィールド識別子を定義する

フィールド識別子 (`fieldid`) は、`typedef` により `FLDID` (FML32 の場合は `FLDID32`) として定義されます。フィールド識別子は、フィールド型とフィールド番号で構成されます。■訳文不要■フィールド番号は、フィールドを識別するための一意な番号です。

フィールド番号は、次の範囲内の番号でなければなりません。

- FML の場合 :1 ～ 8191
- FML32 の場合 :1 ～ 33,554,431

フィールド番号 0 および対応するフィールド識別子 0 は、不正なフィールド識別子 (BADFLDID) を示すために予約されています。フィールド機能を備えた別のソフトウェアを使用して FML を操作する場合は、フィールド番号に関する別の制約が適用されることがあります。

BEA Tuxedo システムは、フィールド番号に関する以下の規則に従っています。

FML のフィールド番号		FML32 のフィールド番号	
予約	**有効**	**予約**	**有効**
1-100	101-8191	1-10,000、30,000,001-33,554,431	10,001-30,000,000

BEA Tuxedo システムでは予約されている番号の使用を強制的に制限していませんが、アプリケーションでは予約番号を使用しないようにしてください。

フィールド識別子とフィールド名のマッピング情報は、フィールド・テーブル・ファイルまたはフィールド・ヘッダ・ファイルに取り込まれます。フィールド・テーブル・ファイルを使用する場合は、この後で説明するマッピング関数を使用して C プログラム内のフィールド名の参照を変換する必要があります。フィールド・ヘッダ・ファイルを使用すると、プログラムのコンパイル時に C プリプロセッサ (UNIX リファレンス・マニュアルの cpp(1) を参照) によってフィールド名からフィールド識別子へのマッピングが行われます。

フィールド・テーブルにアクセスするための関数およびプログラムでは、FLDTBLDIR 環境変数および FIELDTBLS 環境変数を使用して、使用するソース・ディレクトリとフィールド・テーブル・ファイルを指定します。(FML32 で使用される環境変数は、FLDTBLDIR32 および FIELDTBLS32 です。環境変数の設定方法については、3-1 ページの「FML および VIEWS の環境設定」を参照してください。

複数のフィールド・テーブルを使用すると、複数のフィールド・グループに対して別々のディレクトリやファイルを設定できます。ただし、フィールド名およびフィールド番号は、すべてのフィールド・テーブルを通して一意でなければなりません。フィールド・テーブルが C ヘッダ・ファイルに変換されて同じフィールド番号が複数回発生すると、予測できない結果が発生する可能性があるためです。

フィールド・テーブル・ファイルを作成する

フィールド・テーブル・ファイルは、vi などの標準的なテキスト・エディタを使用して作成します。このファイルの形式は、次のとおりです。

- # で始まる行および空白行は無視されます。
- $ で始まる行は、マッピング関数では無視されますが、mkfldhdr で生成されたヘッダ・ファイルに渡されます。このとき $ は除去されます。詳細については、『BEA Tuxedo コマンド・リファレンス』の「mkfldhdr、mkfldhdr32(1)」を参照してください。マッピング関数で行を無視させる機能は、C コメントや what 文字列を、アプリケーション側から C ヘッダ・ファイルに渡すときに便利です。

  注記　ただし COBOL アプリケーションでは、このような行は COBOL COPY ファイルに渡されません。

- 文字列 *base で始まる行には、後続のフィールド番号をオフセットするためのベース値が含まれています。この機能により、関連するフィールドのセットをグループ化し、簡単に番号を付け直すことができます。
- 上記以外のすべての行の形式は、次のとおりです。

  name　　*rel-number*　　*type*　　*flag*　　*comment*

  各項目の説明は次のとおりです。

  - *name* は、フィールドの識別子です。識別子は、C プリプロセッサの識別子の制約に準拠しています。つまり、name に指定できるのは、英数字と下線 (_) のみです。name は、内部で 30 文字以降が切り捨てられるので、name の最初の 30 文字は一意でなければなりません。
  - *rel-number* は、フィールドの相対番号を示す数値です。*base が指定されている場合、この数値を現在の基数に加算すると、フィールドのフィールド番号を取得できます。

- *type* は、フィールドの型を示します。指定できる型は、short、long、float、double、char、string、carray、mbstring、ptr、fml32、または view32 のいずれかです。
- *flag* フィールドは、将来使用するために予約されています。このフィールドにはダッシュ (-) を指定してください。
- *comment* は、フィールドの内容を説明するためのオプションのフィールドです。

項目は、空白文字またはタブで区切ってください。

フィールド・テーブルの例

次のリストは、基数が 500 から 700 にシフトするフィールド・テーブルの例です。基数が 500 のグループの場合、最初のフィールド番号は 501 です。基数が 700 のグループの場合、最初のフィールド番号は 701 です。

コード リスト 4-1 **フィールド・テーブル・ファイル**

```
# following are fields for EMPLOYEE service
# employee ID fields are based at 500
*base 500
#name           rel-number      type           flags   comment
#----           ----------      ----           ------  -------
EMPNAME         1               string         -       emp name
EMPID           2               long           -       emp id
EMPJOB          3               char           -       job type
SRVCDAY         4               carray         -       service date
*base 700
# all address fields are now relative to 700
EMPADDR         1               string         -       street address
EMPCITY         2               string         -       city
EMPSTATE        3               string         -       state
EMPZIP          4               long           -       zip code
```

フィールド名からフィールド識別子にマッピングする

実行時のマッピングは、Fldid() 関数および Fname() 関数を使用して行います。これらの関数は、FLDTBLDIR 環境変数および FIELDTBLS 環境変数で指定されたフィールド・テーブル・ファイルのセットを参照します。FML32 が使用されている場合、Fldid32() 関数と Fname32() 関数は、FLDTBLDIR32 環境変数および FIELDTBLS32 環境変数を参照します。

Fldid は、引数 (フィールド名) を fieldid にマッピングします。次のコードを参照してください。

```
char *name;
extern FLDID Fldid();
FLDID id;
...
id = Fldid(name);
```

Fname は、引数 (fieldid) をフィールド名にマッピングして、Fldid とは逆の処理を行います。次のコードを参照してください。

```
extern char *Fname();
name = Fname(id);
. . .
```

フィールド識別子からフィールド名へのマッピングは、ほとんど使用されません。フィールド識別子がわかっており、その識別子から対応する名前を確認するケースはほとんどないためです。フィールド識別子からフィールド名へのマッピングを行う例は、バッファの出力ルーチンです。バッファの出力ルーチンでは、フィールド化バッファの内容を、理解できる形式で表示する必要があります。

関連項目

- 『BEA Tuxedo FML リファレンス』の「Fldid、Fldid32(3fml)」
- 『BEA Tuxedo FML リファレンス』の「Fname、Fname32(3fml)」

フィールド・テーブルをロードする

Fldid() を最初に呼び出すと、フィールド・テーブル・ファイルがロードされ、必要な検索が実行されます。フィールド・テーブル・ファイルは、その後もロードされたまま残ります。Fldid() が成功すると、引数に対応したフィールド識別子が返されます。失敗すると、Ferror に FBADNAME が設定され、BADFLDID が返されます (FML32 の場合は Ferror32 が設定されます)。

Fldid() でロードしたフィールド・テーブルが占有するデータ領域を回復するには、Fnmid_unload() 関数を呼び出してファイルをすべてアンロードする必要があります。

Fname() 関数は、Fldid() と同じように動作しますが、処理の内容は、フィールド識別子からフィールド名へのマッピングです。ロード対象のフィールド・テーブルは、Fldid() と同じ環境変数を使用して指定しますが、

別のマッピング・テーブルのセットが生成されます。Fname() が成功すると、fldid 引数に指定された名前を含む文字列へのポインタが返されます。失敗すると NULL が返されます。

注記　ポインタは、テーブルがロードされている限り有効です。

Fldid() のエラーは、フィールド・テーブルが見つからないかまたはオープンできない (FFTOPEN)、フィールド・テーブルの構文が間違っている (FFTSYNTAX)、フィールド・テーブル内にノーヒット条件が存在する (FBADFLD)、などが原因で発生します。Fname() で作成したマッピング・テーブルが占有するテーブル領域は、Fidnm_unload() 関数を呼び出して回復できます。

実行時のマッピングを使用するマッピング関数およびその他の FML 関数には、FIELDTBLS 環境変数および FLDTBLDIR 環境変数を正しく設定しておく必要があります。これらの環境変数が設定されていない場合は、デフォルト値が使用されます。これらの環境変数のデフォルト値については、3-1 ページの「FML および VIEWS の環境設定」を参照してください。

関連項目

- 『BEA Tuxedo FML リファレンス』の「Fldid、Fldid32(3fml)」
- 『BEA Tuxedo FML リファレンス』の「Fnmid_unload、Fnmid_unload32(3fml)」
- 『BEA Tuxedo FML リファレンス』の「Fname、Fname32(3fml)」
- 『BEA Tuxedo FML リファレンス』の「Fidnm_unload、Fidnm_unload32(3fml)」

フィールド・テーブルからヘッダ・ファイルに変換する

既に説明したとおり、mkfldhdr (または mkfldhdr32) コマンドを実行すると、フィールド・テーブルが C コンパイラ処理に対応したファイルに変換されます。生成されたヘッダ・ファイルの各行の形式は、次のとおりです。

```
#define fname fieldid
```

fname にはフィールド名を指定し、*fieldid* にはフィールド識別子を指定します。フィールド識別子は、符号化されたフィールド型とフィールド番号で構成されます。フィールド番号は絶対数、つまり base に rel-number を足した数です。生成されたファイルは、C プログラムに組み込むことができます。

4-13 ページの「フィールドを C 構造体および COBOL レコードにマッピングする」で説明するとおり、実行時のマッピング関数を使用する場合、ヘッダ・ファイルを使用する必要はありません。

コンパイル時にファイル名を識別子にマッピングすると、処理を高速化でき、データ領域が少なくて済むという利点があります。逆に、サービス・ルーチンのコンパイル後にフィールド名と識別子のマッピングが変更されると、サービス・ルーチンに伝播されないという欠点があります。このような場合、サービス・ルーチンはコンパイル済みのマッピングを使用します。

mkfldhdr コマンドを実行すると、FIELDTBLS 環境変数で指定された各フィールド・テーブルが、対応するヘッダ・ファイルに変換されます。ヘッダ・ファイル名は、フィールド・テーブル名に .h という接尾辞を付けて指定します。生成されたヘッダ・ファイルは、デフォルトでカレント・ディレクトリに保存されます。別のディレクトリに保存したい場合は、mkfldhdr コマンドで -d オプションを使用して、保存先ディレクトリを指定します。詳細については、『BEA Tuxedo コマンド・リファレンス』の「mkfldhdr、mkfldhdr32(1)」を参照してください。

フィールド・テーブルからヘッダ・ファイルへの変換例

例 1 および 2 は、環境変数を設定して mkfldhdr(1) コマンドを実行する方法を示しています。ここでは、3 つのフィールド・テーブル・ファイル (${FLDTBLDIR}/maskftbl、${FLDTBLDIR}/DBftbl、${FLDTBLDIR}/miscftbl) が処理され、3 つのインクルード・ファイル (maskftbl.h、DBftbl.h、miscftbl.h) がカレント・ディレクトリに生成されます。詳細については、『BEA Tuxedo コマンド・リファレンス』の「mkfldhdr、mkfldhdr32(1)」を参照してください。

例 1

```
FLDTBLDIR=/project/fldtbls
FIELDTBLS=maskftbl,DBftbl,miscftbl
export FLDTBLDIR FIELDTBLS
mkfldhdr
```

例 2

```
FLDTBLDIR32=/project/fldtbls
FIELDTBLS32=maskftbl,DBftbl,miscftbl
export FLDTBLDIR32 FIELDTBLS32
mkfldhdr32
```

例 3

例 3 は例 1 と同じですが、出力ファイル (maskftbl.h、DBftbl.h、miscftbl.h) が ${FLDTBLDIR} で指定されたディレクトリに保存される点だけが異なります。

```
FLDTBLDIR=/project/fldtbls
FIELDTBLS=maskftbl,DBftbl,miscftbl
export FLDTBLDIR FIELDTBLS
mkfldhdr -d${FLDTBLDIR}

mkfldhdr -d${FLDTBLDIR}
```

環境変数をオーバーライドして mkfldhdr を実行する

mkfldhdr のコマンド行でフィールド・テーブルの名前を指定し、環境変数をオーバーライドする (または環境変数を設定しない) ことができます。

ただし、この方法は実行時マッピングの関数に対しては適用できません。実行時マッピングの関数が使用されている場合、FLDTBLDIR がカレント・ディレクトリと見なされ、FIELDTBLS はユーザがコマンド行で指定したパラメータの一覧と見なされます。次の例を参照してください。

```
mkfldhdr myfields
```

このコマンドは、フィールド・テーブル・ファイル myfields をフィールド・ヘッダ・ファイル myfields.h に変換し、そのヘッダ・ファイルをカレント・ディレクトリに格納します。

詳細については、『BEA Tuxedo コマンド・リファレンス』の「mkfldhdr、mkfldhdr32(1)」を参照してください。

フィールドを C 構造体および COBOL レコードにマッピングする

ここでは、次の内容について説明します。

VIEWS 機能とは

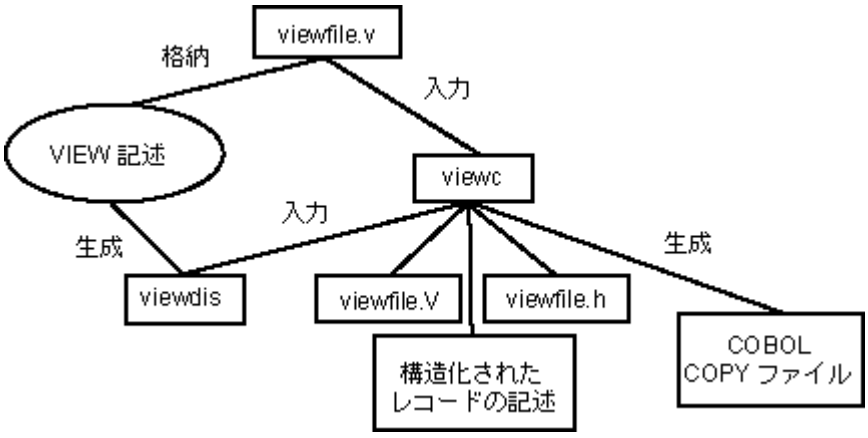
FML VIEWS は、フィールド化バッファと C 構造体、またはフィールド化バッファと COBOL レコードの間でデータを交換するためのメカニズムです。この機能は、時間のかかるデータ操作を、C 関数を使用して C 構造体で行うために用意されています。この方が、FML 関数を使用してフィールド化バッファでデータを操作するより効率的です。また、VIEWS を使用して、COBOL プログラムで FML フィールド化レコードを扱う C プログラムとメッセージを送受信する方法としても使用できます。

この節では、VIEWS を使用してフィールド化バッファと C 構造体をマッピングする方法を説明します。

VIEWS の構造

次の図は、VIEWS の構成要素とそれらの相互関係を示しています。

図 4-1VIEWS 機能の構成要素



VIEW ファイルを作成する

ソース VIEW ファイルは、vi などの任意のテキスト・エディタで作成する標準テキスト・ファイルです。このファイルには、1 つまたは複数のソース VIEW 記述 (フィールドから構造体への実際のマッピング情報) が格納されています。

VIEW コンパイラの最も重要な役割は、オブジェクト VIEW ファイルを生成することです。このファイルには、コンパイル済みのオブジェクト VIEW 記述が含まれています。オブジェクト VIEW ファイルは逆に、VIEW 逆アセンブラ (viewdis または viewdis32) での入力として使用できます。VIEW 逆アセンブラを使用すると、オブジェクト VIEW 記述をソース形式に戻し、内容

を検証したり、編集することができます。詳細については、『BEA Tuxedo コマンド・リファレンス』の「viewdis、viewdis32(1)」を参照してください。

ソース VIEW 記述を作成および編集し、viewdis の出力を編集することができます。コンパイル済みの VIEW 記述 (バイナリ形式) を直接読み取ることはできません。

VIEW ファイルには、VIEW 記述のほかに # または $ で始まるコメント行を格納できます。# で始まる空白行は、VIEW コンパイラによって無視されますが、$ で始まる行は、VIEW コンパイラが生成するヘッダ・ファイルに渡すことができます。この規則によって、C のコメントなどの what の文字列を VIEW コンパイラが生成する C ヘッダ・ファイルに渡すことができます。

注記　この規則は、COBOL には適用されません。したがって、$ で始まる行は COBOL COPY ファイルには渡されません。

VIEW 記述を作成する

ソース VIEW ファイル内の各ソース VIEW 記述は、次の 3 つの部分で構成されています。

- キーワード VIEW で始まり (接尾辞「32」は付かない)、続いて VIEW 記述の名前が指定された行。この名前は、英数字と下線 (_) で構成されます。viewc に指定できる最大文字数は 33 文字ですが、tpalloc(3c) が受け付けるサブタイプの最大文字数が 16 文字であるため、実際に指定できる最大文字数は 16 文字です。
- メンバ記述の一覧
- キーワード END で始まる行

各 VIEW 記述の最初の行は、「VIEW」というキーワードで始まり、続いて VIEW 記述の名前が指定されます。メンバ記述 (マッピング・エントリ) は、C 構造体または COBOL レコードのメンバに関する情報を含む行です。キーワード END で始まる行は、VIEW 記述の最後の行です。

次のリストは、一般的なソース VIEW 記述の内容です。

コード リスト 4-2 VIEW **記述のソース**

```
VIEW vname
  # type   cname   fbname   count   flag   size   null
  # ----   -----   ------   -----   ----   ----   ----
  --------------member descriptions-------------------
  .
  .
  .
  END
```

以下は、リストの説明です。

- *vname* には、VIEW 記述の名前を指定します。この名前は、C 構造体の名前としても使用されるので、有効な C 識別名を指定する必要があります。下線は、COBOL COPY ファイルでは自動的にダッシュ (-) にマッピングされます。
- *type* には、メンバの型を指定します。int、short、long、char、float、double、string、carray、または dec_t です。*type* の値が「-」の場合、デフォルトである *fbname* の値が使用されます。
- *cname* には、構造体メンバの識別子を指定します。これは、C 構造体メンバの名前なので、有効な C 識別名にする必要があります。下線は、COBOL COPY ファイルでは自動的にダッシュ (-) にマッピングされます。
- *fbname* には、フィールド化バッファのフィールド名を指定します。フィールド・テーブル・ファイルにある名前を指定します。
- *count* には、割り当てる要素の数 (このメンバに格納されるオカレンスの最大数) を指定します。FML では 65,535 以下、FML32 では 2,147,483,647 以下の数値を指定します。

- *flag* には、カンマで区切ったオプションのリスト、または「-」(オプションが設定されていない場合) を指定します。詳細については、4-17 ページの「VIEW 記述でのフラグ・オプションを指定する」を参照してください。
- *size* には、タイプが string、carray、または dec_t である場合のメンバのサイズを指定します。その他のフィールド・タイプでは、「-」を指定します。この場合は VIEW コンパイラがサイズを計算します。
  - type が string または carray の場合、*size* の値には、FML では 65,535 以下、FML32 では 2,147,483,647 以下の数値を指定します。
  - type が dec_t の場合、*size* には、カンマで区切って指定した 2 つの数値を指定します。最初の数値は 10 進数のバイト数 (0 より大きく 10 未満) で、2 つ目の数値は小数点の右側の桁数 (0 より大きく、バイト数の 2 倍から 1 を引いた数値未満) です。
- null には、ユーザが定義した NULL 値を指定します。「-」を指定すると、該当するフィールドのデフォルトの NULL 値に設定されます。詳細については、4-21 ページの「VIEWS で NULL 値を使用する」を参照してください。

VIEW 記述でのフラグ・オプションを指定する

以下は、VIEW 記述内のメンバ記述の flag 要素として指定できるオプションです (フラグ・オプションは FML 対応の VIEW にのみ適用されます)。

c

このオプションは、メンバ記述で記述された構造体メンバに加えて、連想カウント・メンバ (ACM: Associated Count Member) と呼ばれる構造体メンバの生成を要求します。

フィールド化バッファから構造体へのデータ転送時には、構造体内の各 ACM は対応する構造体メンバに転送されたオカレンスの数に設定されます。

- ACM の値 0 は、対応する構造体メンバにフィールドが転送されなかったことを示します。

- 正の値は、構造体メンバの配列に実際に転送されたフィールドの数を示します。
- 負の値は、構造体メンバの配列に転送できる数以上のフィールドがバッファ内にあったことを示します。この場合、ACM の絶対値と構造体に転送されないフィールド数は同じです。

構造体メンバの配列からフィールド化バッファへのデータ転送時には、ACM を使用して転送すべき配列要素の数を指定します。たとえば、あるメンバの ACM が N に設定されている場合は、最初の N 個の NULL 以外のフィールドがフィールド化バッファに転送されます。N が配列のサイズより大きい場合、N はデフォルトで配列のサイズに設定されます。どちらの場合も、転送後、ACM はフィールド化バッファに転送される配列のメンバの実際の数に設定されます。

ACM の型は、FML では `short` 型、FML32 では `long` 型として C ヘッダ・ファイルで宣言され、`C_`*`cname`* という名前が生成されます。*`cname`* は、ACM が宣言された `cname` エントリです。たとえば、`parts` という名前のメンバの ACM は、次のように宣言されます。

```
short C_parts;
```

COBOL COPY ファイルでは、`C-`*`cname`* という名前が生成され、型は次のように宣言されます。

- FML の場合 :`PIC S9(4) USAGE COMP-5`
- FML32 の場合 :`PIC S9(9) USAGE COMP-5`

注記　生成された ACM 名と接頭辞 `C_` で始まる名前の構造体メンバが競合する場合があります。VIEW コンパイラは、このような競合を致命的なエラーとして報告します。たとえば、構造体メンバの `C_parts` という名前は、メンバ `parts` に対して生成される ACM 名と競合します。

`F`

このオプションは、構造体からフィールド化バッファへの一方向のマッピングを指定します。このオプションによるメンバのマッピングは、構造体からフィールド化バッファへのデータ転送時のみ有効です。このオプションは、`-n` コマンド行オプションを指定した場合は無視されます。

L

このオプションは、carray 型または string 型のメンバ記述に対してのみ使用され、これらのフィールドが可変長の場合に転送されるバイト数を示します。carray フィールドまたは string フィールドを常に固定長データ項目として使用する場合、このオプションを指定する利点はありません。

L オプションは、carray 型または string 型の構造体メンバに対応する連想長メンバ (ALM: Associated Length Member) を生成します。フィールド化バッファから構造体へのデータ転送時には、ALM は、対応する転送フィールドの長さに設定されます。フィールド化バッファ内のフィールドの長さがマッピングされた構造体メンバに割り当てられた領域を超える場合は、割り当てられたバイト数のみが転送されます。対応する ALM は、フィールド化バッファ項目のサイズに設定されます。したがって、ALM が構造体メンバの配列サイズより大きい場合、フィールド化バッファ情報は転送時に切り捨てられます。

構造体メンバからフィールド化バッファのフィールドにデータを転送する場合、そのフィールドが carray 型であれば、ALM を使用してフィールド化バッファに転送するバイト数を指定します。string 型のフィールドの場合、ALM は転送時に無視されますが、転送後、転送されたバイト数に設定されます。carray フィールドには長さ 0 も指定できます。ALM が 0 の場合は、関連する構造体メンバの値が NULL 値でない限り、フィールド化バッファに長さ 0 のフィールドが転送されます。

ALM の型は、FML では unsigned short 型、FML32 では unsigned long 型として C ヘッダ・ファイルで定義され、L_*cname* という名前が生成されます。*cname* は、ALM が宣言された構造体の名前です。

ALM が宣言されたメンバのオカレンス数が 1 の場合 (またはデフォルトが 1 の場合)、ALM は次のように宣言されます。

```
unsigned short L_cname;
```

一方、オカレンス数が 1 より大きい場合 (N)、ALM は次のように宣言されます。

```
unsigned short L_cname[N];
```

これは ALM 配列と呼ばれます。このような場合は、ALM 配列内の各要素は構造体メンバ (またはフィールド) の対応するオカレンスを参照し

ます。COBOL COPY ファイルでは、FML は PIC 9(4) USAGE COMP-5 型、FML32 は PIC 9(9) USAGE COMP-5 型として宣言され、L-*cname* という名前が生成されます。メンバが複数回発生する場合は、COBOL の OCCURS 句を使用して複数のオカレンスを定義します。

注記　生成された ALM 名と接頭辞 L_ で始まる名前の構造体メンバが競合する場合があります。VIEW コンパイラは、このような競合を致命的なエラーとして報告します。たとえば、構造体メンバの L_parts という名前は、メンバ parts に対して生成される ALM 名と競合します。

N

ゼロ方向マッピングを指定します。C 構造体にフィールド化バッファはマッピングされません。このオプションは、C 構造体または COBOL レコードに充填文字を割り当てるときに使用することができます。このオプションは、-n コマンド行オプションが指定された場合は無視されます。

P

このオプションは、string 型および carray 型の構造体メンバの NULL 値として VIEW が解釈する内容を指定します。このオプションを指定しない場合は、構造体メンバの値がユーザ指定の NULL 値と等しいと (ただし、後続の NULL 文字は考慮に入れない)、構造体メンバが NULL になります。

一方、このオプションを指定した場合は、ユーザ指定の NULL 値と構造体メンバの値が等しく、しかも文字列の長さが最大文字数に達していると (ただし、後続の NULL 文字は考慮に入れない)、構造体メンバが NULL になります。

C 構造体または COBOL レコードからフィールド化バッファへデータを転送する場合、値が NULL のメンバは転送先のバッファに送信されません。たとえば、構造体のメンバ TEST が carray[25] 型で、ユーザ定義の NULL 値「abcde」が指定されているとします。P オプションを設定しない場合、最初の 5 文字が順に a、b、c、d、e であれば、TEST は NULL と見なされます。P オプションを指定した場合、最初の 4 文字が順に a、b、c、d で、かつ carray の残りの文字がすべて「e」であれば (e が 21 個)、TEST は NULL と見なされます。

このオプションは、-n コマンド行オプションを指定した場合は無視されます。

S

このオプションは、フィールド化バッファから構造体への一方向のマッピングを指定します。このオプションを指定したメンバのマッピングは、フィールド化バッファから構造体へのデータ転送時のみ有効です。このオプションは、-n コマンド行オプションを指定した場合は無視されます。

VIEWS で NULL 値を使用する

VIEWS の NULL 値は、C 構造体または COBOL レコードのメンバが空であることを示すために使用されます。NULL 値はデフォルト値として提供されていますが、独自に定義することもできます。

デフォルトの NULL 値は、数値型の場合はすべて 0 (dec_t の場合は 0.0)、char 型の場合は、“\0”、string 型と carray 型の場合は “ ”。

NULL 値にエスケープ定数を指定することもできます。VIEW コンパイラで認識されるエスケープ定数は、*ddd* (*d* は 8 進数)、\0、\n、\t、\v、\b、\r、\f、\\、\′、および \″ です。

string 型、carray 型、および char 型の NULL 値は、二重引用符または一重引用符で囲みます。VIEW コンパイラは、ユーザ定義の NULL 値内の、エスケープされていない引用符は受け付けません。

また、要素の値がその要素の NULL 値と同じ場合、その要素は NULL になります。ただし、例外として以下の場合があります。

- 構造体メンバに P オプションが設定されており、構造体メンバの型が string 型または carray 型である場合。P オプションの詳細については、前の節を参照してください。
- メンバの型が string 型の場合。メンバの値は NULL 値と同じ文字列でなければなりません。
- メンバの型が carray 型であり、NULL 値の長さが N の場合。文字配列内の最初の N 個の文字は NULL 値と同じでなければなりません。

VIEW メンバ記述の NULL フィールドにキーワード「NONE」を指定することもできます。このキーワードは、そのメンバの NULL 値がないことを示します。

string 型および文字配列 (carray) 型のメンバのデフォルトの最大サイズは、2660 文字です。

注記　string 型のメンバは通常「\0」で終了するため、ユーザ定義の NULL 値の最後の文字として「\0」を指定する必要はありません。

VIEW ファイルをコンパイルする

FML では viewc、FML32 では viewc32 が VIEW コンパイラ・プログラムです。これらのプログラムは、ソース VIEW ファイルを取り込み、オブジェクト VIEW ファイルを生成します。生成されたオブジェクト VIEW ファイルは実行時に解釈され、データの実際のマッピングに影響を与えます。実行時には、viewc に対して C コンパイラを使用する必要があります。コマンド行は、次のようになります。

viewc [-n] [-d *viewdir*] [-C] viewfile [viewfile . .]

このコマンド行の *viewfile* は、ソース VIEW 記述を含むソース VIEW ファイルの名前です。コマンド行には、1 つまたは複数の *viewfile* を指定できます。

-C オプションを指定すると、*viewfile* で定義した各 VIEW に対して COBOL COPY ファイルが 1 つ作成されます。これらのコピー・ファイルはカレント・ディレクトリに作成されます。

-n オプションを使用すると、FML バッファにマッピングしない C 構造体または COBOL レコードの VIEW 記述ファイルをコンパイルできます。

デフォルトでは、*viewfile* 内のすべての VIEW がコンパイルされ、複数のファイルが生成されます。つまり、VIEW ファイルごとにオブジェクト VIEW ファイル (接尾辞「v」) とヘッダ・ファイル (接尾辞「h」) が作成されます。VIEWS の構成要素については、4-14 ページの「VIEWS 機能の構成要素」を参照してください。

オブジェクト VIEW ファイルの名前は viewfile.v となります。オブジェクト VIEW ファイルは、カレント・ディレクトリに作成されます。-d オプションを指定して別のディレクトリを指定することもできます。ヘッダ・ファイルは、カレント・ディレクトリ内に作成されます。

注記　Windows のように、大文字と小文字を区別しないオペレーティング・システムの場合、オブジェクト VIEW ファイルには .vv という接尾辞が付きます。

詳細については、『BEA Tuxedo コマンド・リファレンス』の「viewc、viewc32(1)」を参照してください。

viewc を使ってコンパイルしたヘッダ・ファイルを使用する

VIEW コンパイラ (viewc) を使用して作成したヘッダ・ファイルを任意の C アプリケーション・プログラム内で使用すると、VIEW で記述した C 構造体を宣言できます。たとえば、次のような VIEW 記述があるとします。

```
VIEW test
#TYPE   CNAME    FBNAME    COUNT    FLAG    SIZE     NULL
int     empid    EMPID       1        -       -      -1
float   salary   EMPPAY      1        -       -      0
long    phone    EMPPHONE    4        -       -      0
string  name     EMPNAME     1        -       32     "NO NAME"
END
```

この VIEW 記述の C ヘッダ・ファイルは、次のようになります。

```
struct test {
 long    empid;                 /* null=-1 */
 float   salary;                /* null=0.000000 */
 long    phone[4];              /* null=0 */
 char    name[32];              /* null="NO NAME" */
};
```

詳細については、『BEA Tuxedo コマンド・リファレンス』の「viewc、viewc32(1)」を参照してください。

VIEW コンパイラを使って作成した COBOL COPY ファイルを使用する

-c オプションを使用して VIEW コンパイラで作成した COBOL COPY ファイルを任意の COBOL アプリケーション・プログラムで使用すると、VIEW で記述した COBOL レコードを宣言できます。たとえば、前の節に示した VIEW 記述の COBOL COPY ファイルは、TEST.cbl ファイルでは次のようになります。

```
*       VIEWFILE: "test.v"
*       VIEWNAME: "test"
05 EMPID                   PIC S9(9) USAGE IS COMP-5.
05 SALARY                  USAGE IS COMP-1.
05 PHONE OCCURS 4 TIMES    PIC S9(9) USAGE IS COMP-5.
05 NAME                    PIC X(32).
```

COPY ファイルの名前は、VIEW コンパイラによって自動的に大文字に変換されます。COPY ファイルは、次のように COBOL プログラムに組み込まれます。

```
01 MYREC COPY TEST.
```

COPY ファイルの出力の詳細については、『COBOL を使用した BEA Tuxedo アプリケーションのプログラミング』を参照してください。

コンパイル後に VIEW ファイルの情報を表示する

VIEW 逆アセンブラである viewdis は、VIEW コンパイラで生成されたオブジェクト VIEW ファイルを逆処理し、ソース VIEW ファイルの形式で VIEW 情報を表示します。また、対応する構造体メンバのオフセットも表示します。

この形式の情報を参照できると、オブジェクト VIEW 記述が正確かどうかを検証するのに役立ちます。

VIEW 逆アセンブラを実行するには、次のコマンドを入力します。

```
viewdis objviewfile . . .
```

デフォルトでは、カレント・ディレクトリ内の *objviewfile* が逆アセンブルされます。このファイルをカレント・ディレクトリで検索できないと、エラー・メッセージが表示されます。コマンド行には、1 つまたは複数のオブジェクト VIEW ファイルを指定できます。

viewdis の出力は、元のソース VIEW 記述と同様であり、編集したり viewc に再入力できます。viewdis の出力に表示される行の順序は、元のソース VIEW 記述の順序と異なる場合がありますが、順序が違ってもオブジェクト VIEW ファイルの正確性とは関係ありません。

詳細については、『BEA Tuxedo コマンド・リファレンス』の「viewdis、viewdis32(1)」を参照してください。

5 フィールド操作関数

ここでは、次の内容について説明します。

この章について

この章では、4-1 ページの「フィールドの定義と使用」で説明した実行時のマッピング関数を除く、すべての FML 関数および FML VIEWS 関数について説明します。

COBOL プログラムでは、FML 関数を直接使用できません。まず、`FINIT` というプロシージャを使用して、FML データを受信するためにレコードを初期化し、次に `FVSTOF` および `FVFTOS` のプロシージャを使用して COBOL レコードと FML バッファ間の変換を行います。これらのプロシージャの詳細については、『COBOL を使用した BEA Tuxedo アプリケーションのプログラミング』を参照してください。ここでは、COBOL インターフェイスについては説明しません。

FML と VIEWS:16 ビット・インターフェイスと 32 ビット・インターフェイス

FML は、2 種類あります。従来の FML インターフェイスは、フィールド長に 16 ビットの値を使用し、フィールドを識別する情報を格納します。したがって、FML16 と呼ばれます。FML16 では、8191 個の一意のフィールド、個々のフィールド長は 64K バイト、フィールド化バッファの総容量は 64K に制限されます。このインターフェイスの定義、型、および関数のプロトタイプは `fml.h` 内にあり、このファイルは FML16 インターフェイスを使用するアプリケーション・プログラムに組み込む必要があります。各関数は、`-lfml` にあります。

もう 1 つのインターフェイスである FML32 は、フィールド長および識別子に 32 ビットの値を使用します。FML32 では、約 3 千万のフィールドを使用でき、約 20 GB のフィールド長およびバッファ長を使用できます。FML32

の定義、型、および関数のプロトタイプは、fml32.h 内にあります。各関数は、-lfml32 にあります。FML32 の定義、型、および関数名には、すべて接尾辞「32」が付きます (MAXFBLEN32、FBFR32、FLDID32、FLDLEN32、F_OVHD32、Fchg32、エラー・コード Ferror32 など)。環境変数にも接尾辞「32」が付きます (FLDTBLDIR32、FIELDTBLS32、VIEWFILES32、VIEWDIR32 など)。FML32 では、フィールド化バッファのポインタは「FBFR32 *」型、フィールド長は FLDLEN32 型、フィールドのオカレンス数は FLDOCC32 型です。FML32 バッファには、デフォルトで 4 バイトの調整が必要です。

正しく記述された FML16 アプリケーションは、簡単に FML32 インターフェイスに変更できます。FML 関数の呼び出しに使用する変数は、すべて適切な typedef (FLDID、FLDLEN、および FLDOCC) により定義されている必要があります。FML の型付きバッファに対して tpalloc(3c) を呼び出すときは、FML の代わりに FMLTYPE で定義する必要があります。アプリケーションのソース・コードには、fml.h の代わりに fml32.h を指定し、fml1632.h を組み込むことで 32 ビットの関数を使用できるようになります。fml1632.h には、すべての 16 ビットの型定義を 32 ビット版に変換したり、16 ビットの関数やマクロを 32 ビット版に変換するマクロが含まれています。

FML32 のフィールド化バッファを FML16 のフィールド化バッファに変換する関数や、その逆の変換を行う関数も提供されています。

```
#include “fml.h”
#include “fml32.h”
int
F32to16(FBFR *dest, FBFR32 *src)
int
F16to32(FBFR32 *dest, FBFR *src)
```

F32to16 は、32 ビットの FML バッファを 16 ビットの FML バッファに変換します。これは、フィールド対フィールドのバッファの変換を行い、フィールド化バッファのインデックスを作成することによって行ないます。FLDID32 から FLDID を生成し、フィールド値 (string 型および carray 型フィールドではフィールド長も含む) をコピーすると、フィールドは変換されます。

dest は、変換後のフィールド化バッファを示すポインタ (宛先バッファ) であり、*src* は、変換元のフィールド化バッファを示すポインタ (ソース・バッファ) です。ソース・バッファは変更されません。

これらの関数は、領域の不足により失敗する可能性があります。操作を完了するには、十分な追加領域を割り当ててから関数を再度発行します。F16to32 は、16 ビットの FML バッファを 32 ビットの FML バッファに変換します。この関数は、fml32 ライブラリまたは共用オブジェクトに格納されており、エラーが発生すると Ferror32 が設定されます。F32to16 は、fml ライブラリまたは共用オブジェクトに格納されており、エラーが発生すると Ferror が設定されます。これらの関数を使用するためには、fml.h および fml32.h の両方をインクルードする必要があることに注意してください。同じファイルに、fml1632.h をインクルードする必要はありません。

埋め込み型バッファのフィールド型 (FLD_PTR、FLD_FML32、および FLD_VIEW32) は、FML32 でのみサポートされています。FLD_PTR 型、FLD_FML32 型、FLD_MBSTRING 型、または FLD_VIEW32 型のフィールドを含むバッファを使用すると、F32to16 が失敗し、FBADFLD エラーが返されます。これらの関数に対して F16to32 を呼び出しても、何も起こりません。

注記　以降の節では、16 ビットの関数のみ説明し、対応する FML32 および VIEW32 関数は説明しません。

FML 関数のパラメータの定義

FML 関数のパラメータ仕様をわかりやすくするため、パラメータは、規則的な順序で指定されます。FML のパラメータは、次の順序で指定されます。

1. フィールド化バッファに対するポインタ (FBFR) を必要とする関数の場合は、このポインタを最初に指定します。フィールド化バッファに対するポインタが 2 つ必要な関数 (転送関数など) の場合は、最初に宛先バッファに対するポインタを指定し、次にソース・バッファに対するポインタを指定します。フィールド化バッファに対するポインタは、short 型の境界に調整した領域を指す必要があります (そうでない場合は、Ferror に FALIGNERR が設定されたエラーが返されます)。また、領域はフィールド化バッファでなければなりません (そうでない場合は、Ferror に FNOTFLD が設定されたエラーが返されます)。

2. 入出力関数の場合は、最初にストリームに対するポインタを指定し、次にフィールド化バッファに対するポインタを指定します。
3. フィールド識別子を必要とする関数の場合は、次にフィールド識別子 (FLDID 型) を指定します (Fnext の場合は、次にフィールド識別子に対するポインタを指定します)。
4. フィールド・オカレンス (FLDOCC 型) を必要とする関数の場合は、次にフィールド・オカレンスを指定します (Fnext の場合は、次にオカレンス番号に対するポインタを指定します)。
5. フィールド値を受け渡す関数の場合は、次にフィールド値の先頭に対するポインタを指定します (このポインタは、文字ポインタとして定義しますが、別の型のポインタからキャストすることもできます)。
6. 文字配列 (carray、mbstring) 型フィールドを処理する関数にフィールド値を渡す場合は、次にフィールドの長さを決定するパラメータ (FLDLEN 型) を指定します。フィールド値を検索する関数の場合は、検索対象のバッファの長さに対するポインタを関数に渡す必要があります。このパラメータは、検索対象の値の長さに設定されます。
7. いくつかの関数では、上記の規則には当てはまらない特別なパラメータが必要です。これらの特別なパラメータは、上記のパラメータの後に指定します。詳細については、各関数の説明部分を参照してください。
8. 以下は、フィールドの型ごとに定義されている NULL 値です。
   - short 型と long 型の場合は 0
   - float 型と double 型の場合は 0.0
   - string 型の場合は \0 (長さ 1 バイト)
   - carray 型または mbstring 型の場合は長さ 0 の文字列

フィールド識別子をマッピングする関数

以下の関数を使用すると、プログラムの実行時にフィールド・テーブルまたはフィールド識別子を照会し、フィールドに関する情報を取得できます。

Fldid

Fldid は、指定された有効なフィールド名のフィールド識別子を返し、フィールド名 / フィールド識別子のマッピング・テーブルがない場合は、そのテーブルをフィールド・テーブル・ファイルからロードします。

```
FLDID
Fldid(char *name)
```

name は有効なフィールド名です。

マッピング・テーブルが使用するメモリ領域は、Fnmid_unload、Fnmid_unload32(3fml) 関数を使用して解放することができます。ただし、これらのテーブルは、Fname 関数でロードされ、使用されるテーブルとは異なります。

詳細については、『BEA Tuxedo FML リファレンス』の「Fldid、Fldid32(3fml)」を参照してください。

Fname

Fname は、指定された有効なフィールド識別子のフィールド名を返し、フィールド識別子 / フィールド名のマッピング・テーブルがない場合は、そのテーブルをフィールド・テーブル・ファイルからロードします。

```
char *
Fname(FLDID fieldid)
```

fieldid は、有効なフィールド識別子です。

マッピング・テーブルが使用するメモリ領域は、Fnmid_unload、Fnmid_unload32(3fml) 関数を使用して解放することができます。ただし、これらのテーブルは、Fldid 関数によってロードおよび使用されるテーブルとは別のテーブルです。詳細については、『BEA Tuxedo FML リファレンス』を参照してください。

詳細については、『BEA Tuxedo FML リファレンス』の「Fname、Fname32(3fml)」を参照してください。

Fldno

Fldno は、指定されたフィールド識別子からフィールド番号を抽出します。

```
FLDOCC
Fldno(FLDID fieldid)
```

fieldid は、有効なフィールド識別子です。

詳細については、『BEA Tuxedo FML リファレンス』の「Fldno、Fldno32(3fml)」を参照してください。

Fldtype

Fldtype は、指定されたフィールド識別子からフィールドの型 (fml.h で定義された整数) を抽出します。

```
int
Fldtype(FLDID fieldid)
```

fieldid は、有効なフィールド識別子です。

次の表は、Fldtype が返す値とその意味です。

表 5-1Fldtype が返すフィールド型

戻り値	説明	fml.h または fml32.h でのフィールド型名
0	short 型整数	FLD_SHORT
1	long 型整数	FLD_LONG
2	文字	FLD_CHAR
3	単精度浮動小数点	FLD_FLOAT
4	倍精度浮動小数点	FLD_DOUBLE
5	NULL 終了文字列	FLD_STRING
6	文字配列	FLD_CARRAY
9	ポインタ	FLD_PTR
10	埋め込み型の FML32 バッファ	FLD_FML32
11	埋め込み型の VIEW32 バッファ	FLD_VIEW32
12	マルチバイト文字配列	FLD_MBSTRING

詳細については、『BEA Tuxedo FML リファレンス』の「Fldtype、Fldtype32(3fml)」を参照してください。

Ftype

Ftype は、フィールド識別子で指定されたフィールドの型名を含む文字列に対するポインタを返します。

```
char *
Ftype(FLDID fieldid)
```

fieldid は、有効なフィールド識別子です。たとえば、次のコードは、short 型、long 型、char 型、float 型、double 型、string 型、carray 型、mbstring 型、FLD_PTR 型、FLD_FML32 型、FLD_VIEW32 型のいずれか 1 つの文字列に対するポインタを返します。

```
char *typename
. . .
typename = Ftype(fieldid);
```

詳細については、『BEA Tuxedo FML リファレンス』の「Ftype、Ftype32(3fml)」を参照してください。

Fmkfldid

アプリケーションの作成機能の一部として、またはフィールド識別子を再作成するため、型の種類とフィールド番号からフィールド識別子を作成しておくと便利です。Fmkfldid は、この機能を提供します。

```
FLDID
Fmkfldid(int type, FLDID num)
```

以下はパラメータの説明です。

- *type* は、有効な型、つまり整数です。詳細については、5-7 ページの「Fldtype」を参照してください。
- *num* は、フィールド番号です。既存のフィールドとの混同を避けるため、未使用のフィールド番号を指定する必要があります。

詳細については、『BEA Tuxedo FML リファレンス』の「Fmkfldid、Fmkfldid32(3fml)」を参照してください。

バッファの割り当ておよび初期化を行う関数

ここでは、スタンドアロンの FML プログラムをコーディングするための関数を説明します。BEA Tuxedo の ATMI 関数を使用している場合に、メッセージ・バッファを割り当てたり、割り当てを解除するには、Falloc、Falloc32(3fml)、Frealloc、Frealloc32(3fml)、Ffree、Ffree32(3fml) などの FML 関数の代わりに、tpalloc(3c)、tprealloc(3c)、tpfree(3c) などの ATMI 関数を呼び出さなければなりません。

大半の FML 関数では、引数としてフィールド化バッファに対するポインタが必要です。このようなポインタを宣言するには、次の例のように、typedef 宣言をした FBFR を使用できます。

```
FBFR *fbfr;
```

この節では、変数 *fbfr* をフィールド化バッファに対するポインタとして使用します。フィールド化バッファ自体は宣言できません。宣言できるのは、フィールド化バッファに対するポインタだけです。

サーバは、FML バッファを含む要求を受け取ると、受け取った FML バッファと埋め込み型の VIEW または FLD_PTR フィールドが参照するバッファに対して領域を割り当てます。新しい FML バッファに対するポインタが、ユーザ記述コードに渡されます。サーバの処理が終了したら、メッセージ受信時に割り当てたバッファはすべて破棄しなければなりません。BEA Tuxedo システムによって FML バッファとすべての補助バッファが調べられ、参照が見つかったバッファはすべて削除されます。サーバ・コードをコーディングする場合は、以下の状況を認識しておく必要があります。

- サーバによって割り当てられたバッファを参照するように VIEW またはポインタ・フィールドを追加、変更、または更新した場合、新たに割り当てられたバッファは、tpreturn(3c) または tpforward(3c) 関数の呼び出しによって開始されるクリーンアップ中に削除されます。
- バッファが FML バッファによって参照されないようにフィールドを変更、更新、または削除した場合は、ユーザ記述コードで tpfree(3c) 関数を使って、そのバッファを明示的に解放しなければなりません。バッ

ファが明示的に解放されていない場合は、サーバ・プロセスでメモリ・リークが発生します。

- tpreturn(3c) を呼び出す代わりに、ユーザ記述コードを使用して別のバッファを割り当て、返すことができます。この場合、tpreturn() に渡された FML バッファは解放されますが、FLD_PTR フィールドまたは FLD_VIEW32 フィールドによって参照されるバッファは解放されません。

この節では、フィールド化バッファの領域を確保するための関数について説明します。まず、指定されたバッファがフィールド化バッファであるかどうかを判別するための関数を説明します。

Fielded

Fielded (または Fielded32) を使用すると、指定されたバッファがフィールド化されているかどうかをテストできます。

```
int
Fielded(FBFR *fbfr)
```

Fielded32 は 32 ビットの FML と共に使用します。

Fielded を使用すると、バッファがフィールド化されている場合は true (1) が返されます。バッファがフィールド化されていない場合は false (0) が返されます。この場合、Ferror は設定されません。

詳細については、『BEA Tuxedo FML リファレンス』の「Fielded、Fielded32(3fml)」を参照してください。

Fneeded

フィールド化バッファに割り当てるメモリ領域のサイズは、フィールド化バッファに指定できるフィールドの最大数と、すべてのフィールド値に必要な領域の合計サイズによって異なります。Fneeded を使用すると、フィールド化バッファに必要な領域 (バイト単位) を判別できます。この関数には引数として、フィールド数とすべてのフィールド値に必要な領域 (バイト単位) を指定します。

```
long
Fneeded(FLDOCC F, FLDLEN V)
```

以下はパラメータの説明です。

- *F* は、フィールド数です。
- *V* はフィールド値に必要な領域 (バイト単位) です。

フィールド値に必要な領域は、各フィールド値が標準的な構造で格納された場合の領域を見積もることにより、算出できます。たとえば、long 型の値が long 型として格納されると、4 バイトの領域が必要です。可変長フィールドの場合は、フィールドに必要な領域の平均値を見積もります。Fneeded で算出される領域には、フィールドごとの固定オーバーヘッドが含まれます。つまり、フィールド値に必要な領域には、オーバーヘッド分が加算されます。

Fneeded を使用して必要な領域を見積もったら、malloc(3) を使用して必要なバイト数を割り当て、割り当てたメモリ領域に対するポインタを設定できます。たとえば、次のコードでは、フィールド化バッファに対し、25 個のフィールドと 300 バイトの値を格納できるだけの領域を割り当てています。

```
#define NF 25
#define NV 300
extern char *malloc;
. . .
  if((fbfr = (FBFR *)malloc(Fneeded(NF, NV))) == NULL)
      F_error("pgm_name");   /* バッファを割り当てる領域がない */
```

ただし、このコードで割り当てられたメモリ領域は、まだフィールド化バッファではありません。この領域は、Finit を使用して初期化する必要があります。

詳細については、『BEA Tuxedo FML リファレンス』の「Fneeded、Fneeded32(3fml)」を参照してください。

Fvneeded

Fvneeded 関数を使用すると、VIEW バッファに必要な領域 (バイト単位) を判別できます。この関数には、引数として VIEW の名前に対するポインタを指定します。

```
long
Fvneeded(char *subtype)
```

Fvneeded 関数は、VIEW のサイズをバイト数で返します。

詳細については、『BEA Tuxedo FML リファレンス』の「Fvneeded、Fvneeded32(3fml)」を参照してください。

Finit

Finit は、割り当てられたメモリ領域をフィールド化バッファとして初期化します。

```
int
Finit(FBFR *fbfr, FLDLEN buflen)
```

以下はパラメータの説明です。

- *fbfr* は、初期化されていないフィールド化バッファに対するポインタです。
- *buflen* は、バッファの長さ (バイト単位) です。

Fneeded の例で割り当てたメモリ領域を初期化するために Finit を呼び出す場合は、以下の形式を使用します。

```
Finit(fbfr, Fneeded(NF, NV));
```

この呼び出しの結果、fbfr は、初期化された空のフィールド化バッファを指します。Fneeded(NF, NV) によって割り当てられた最大バイト数から、fml.h で定義された F_OVHD 分を差し引いたバイト数を、バッファのフィールド領域として使用できます。

注記　ただし、malloc(3) (前の節を参照) への呼び出しと Finit への呼び出しでは、同じ値を使用する必要があります。

詳細については、『BEA Tuxedo FML リファレンス』の「Finit、Finit32(3fml)」を参照してください。

Falloc

Fneeded、malloc(3)、および Finit に対する呼び出しは、Falloc に対する 1 回の呼び出しで置き換えることができます。この関数を使用すると、必要な領域を割り当て、バッファを初期化できます。

```
FBFR *
Falloc(FLDOCC F, FLDLEN V)
```

以下はパラメータの説明です。

- *F* は、フィールド数です。
- *V* は、フィールド値に必要な領域 (バイト単位) です。

前の 3 つの節で説明した、Fneeded、malloc()、および Finit と同じ機能を実行する Falloc への呼び出しは、次のようにコーディングします。

```
extern FBFR *Falloc;
. . .
 if((fbfr = Falloc(NF, NV)) == NULL)
      F_error("pgm_name");   /* バッファを割り当てることができなかった */
```

Falloc (または、Fneeded、malloc(3) および Finit) によって割り当てられた領域を解放するには、Ffree を使用します。『BEA Tuxedo FML リファレンス』の「Ffree、Ffree32(3fml)」を参照してください。

詳細については、『BEA Tuxedo FML リファレンス』の「Falloc、Falloc32(3fml)」を参照してください。

Ffree

Ffree は、フィールド化バッファとして割り当てられたメモリ領域を解放します。Ffree32 は、FLD_PTR フィールドのポインタによって参照されるメモリ領域を解放しません。

```
int
Ffree(FBFR *fbfr)
```

fbfr は、フィールド化バッファに対するポインタです。次の例を参照してください。

```
#include  <fml.h>
. . .
if(Ffree(fbfr) < 0)
      F_error("pgm_name");      /* フィールド化バッファではない */
```

free(3) より Ffree の使用をお勧めします。free(3) ではフィールド化バッファを無効にできませんが、Ffree を使用するとフィールド化バッファを無効にできます。フィールド化バッファの無効化が必要なのは、malloc(3) が解放されたメモリをクリアせずに再利用するためです。free(3) を使用すると、malloc が無効化されていないフィールド化バッファを返す可能性があります。

フィールド化バッファの領域を直接確保することもできます。フィールド化バッファは、short 型境界に調整させて開始する必要があります。バッファには、少なくとも F_OVHD 分のバイト (fml.h で定義) を割り当てる必要があります。割り当てないと、Finit からエラーが返されます。

以下のコードは、前の節の例と似ていますが、Fneeded はマクロではないため、静的バッファのサイズ指定には使用できません。

```
/* 最初の行でバッファを調整している */
static short buffer[500/sizeof(short)];
FBFR *fbfr=(FBFR *)buffer;
. . .
Finit(fbfr, 500);
```

以下のようなコードは入力しないでください。

```
FBFR badfbfr;
. . .
Finit(&badfbfr, Fneeded(NF, NV));
```

このコードは間違いです。FBFR の構造体がユーザのヘッダ・ファイルで定義されていません。したがって、コンパイル時にエラーが発生します。

詳細については、『BEA Tuxedo FML リファレンス』の「Ffree、Ffree32(3fml)」を参照してください。

Fsizeof

Fsizeof は、バイト単位でフィールド化バッファのサイズを返します。

```
long
Fsizeof(FBFR *fbfr)
```

fbfr は、フィールド化バッファに対するポインタです。たとえば、次のコードでは、Fsizeof は、フィールド化バッファの当初の割り当て時に Fneeded が返した数と同じ数を返します。

```
long bytes;
. . .
bytes = Fsizeof(fbfr);
```

詳細については、『BEA Tuxedo FML リファレンス』の「Fsizeof、Fsizeof32(3fml)」を参照してください。

Funused

Funused を使用すると、フィールド化バッファの領域のうち、追加データを格納するために使用できる領域を判別できます。

```
long
Funused(FBFR *fbfr)
```

fbfr は、フィールド化バッファに対するポインタです。次の例を参照してください。

```
long unused;
. . .
unused = Funused(fbfr);
```

Funused は、空き領域のバイト数のみを示し、バッファ内での空き領域の位置は示しません。

詳細については、『BEA Tuxedo FML リファレンス』の「Funused、Funused32(3fml)」を参照してください。

Fused

Fused を使用すると、フィールド化バッファ内でデータおよびオーバーヘッド用に使用されている領域を判別できます。

```
long
Fused(FBFR *fbfr)
```

fbfr は、フィールド化バッファに対するポインタです。次の例を参照してください。

```
long used;
. . .
used = Fused(fbfr);
```

Fused は、使用済み領域のバイト数のみを示し、バッファ内での位置は示しません。

詳細については、『BEA Tuxedo FML リファレンス』の「Fused、Fused32(3fml)」を参照してください。

Frealloc

この関数を使用すると、Falloc を呼び出して既に領域を割り当てたバッファのサイズを変更できます。

tpalloc(3c) で領域を割り当てた場合は、tprealloc(3c) を呼び出して領域を再び割り当てます。バッファのサイズ変更機能は、新しいフィールド値を追加したために空き領域が不足した場合などに役立ちます。このような場合は、単に Frealloc を呼び出すだけでバッファのサイズを増やすことができます。また、Frealloc を呼び出してバッファのサイズを小さくすることもできます。

```
FBFR *
Frealloc(FBFR *fbfr, FLDOCC nf, FLDLEN nv)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *nf* は、新しいフィールド数 (または 0) です。

■ *nv* は、新しいフィールド値の領域 (バイト単位) です。

次の例を参照してください。

```
FBFR *newfbfr;
. . .
if((newfbfr = Frealloc(fbfr, NF+5, NV+300)) == NULL)
        F_error(“pgm_name”);      /* 領域を再割り当てできなかった */
else
        fbfr = newfbfr;           /* 新しいポインタを設定する */
```

この例では、アプリケーション側で、既に割り当てられたフィールド数と値の領域のバイト数を認識している必要があります。Frealloc の引数 (および対応する realloc(3)) は絶対値であり、増分値ではありません。領域の割り当てを複数回行う必要がある場合、このコード例は正しく動作しません。

以下の例では、割り当てられた領域を増分する別の方法を示します。

```
/* バッファがなくなった時にサイズを増やす */
#define INCR    400
FBFR *newfbfr;
.. .
if((newfbfr = Frealloc(fbfr, 0, Fsizeof(fbfr)+INCR)) == NULL)
      F_error(“pgm_name”);        /* 領域を再割り当てできなかった */
else
      fbfr = newfbfr;            /* 新しいポインタを設定する */
```

この例では、バッファが最後に初期化されたときのフィールド数や値の領域のサイズを知っておく必要はありません。したがって、サイズを増やす最も簡単な方法は、現在のサイズに増分値を加えたサイズを値の領域として使用することです。上の例は、必要に応じて何回でも実行することができます。過去の実行内容や値を覚えておく必要はありません。Frealloc を呼び出した後で Finit を呼び出す必要はありません。

Frealloc に対する呼び出しで要求した追加の領域が古いバッファと隣接している場合は、上の例の newfbfr と fbfr は同一になります。ただし、プログラムが正しく実行されるようにするため、newfbfr を宣言しておき、新しい値または NULL 値が返されるのを防ぐ必要があります。Frealloc が失敗した場合、fbfr は再使用しないでください。

注記　バッファ・サイズは、バッファ内で現在使用中のバイト数までしか縮小できません。

詳細については、『BEA Tuxedo FML リファレンス』の「Frealloc、Frealloc32(3fml)」を参照してください。

フィールド化バッファを移動する関数

フィールド化バッファの位置に関する唯一の制約は、フィールド化バッファを short 型境界に調整しなければならないことです。この制約がなければ、フィールド化バッファをメモリ内で自由に移動できます。

Fmove

src がフィールド化バッファを指し、*dest* がフィールド化バッファを保持できるサイズの記憶領域を指す場合は、以下のコードを使用してフィールド化バッファを移動できます。

```
FBFR *src;
char *dest;
. . .
memcpy(dest, src, Fsizeof(src));
```

C の実行時メモリ管理関数の 1 つである memcpy は、指定したバイト数のバッファ (3 つ目の引数で指定) を、指定した領域 (2 つ目の引数で指定) から別の領域 (1 つ目の引数で指定) に移動します。

memcpy を使用するとフィールド化バッファをコピーできますが、コピー先のバッファは、コピー元と同じになります。たとえば、コピー先の空き領域のバイト数は、コピー元の空き領域のバイト数と同じです。

Fmove は、memcpy と同じように動作します。ただし、明示的に長さを指定する必要はありません (自動的に算出されます)。

```
int
Fmove(char *dest, FBFR *src)
```

以下はパラメータの説明です。

- *dest* は、宛先バッファに対するポインタです。

- *src* は、ソース・フィールド化バッファに対するポインタです。

たとえば、次のコードでは、Fmove は、ソース・バッファがフィールド化バッファかどうかを確認しますが、このバッファの変更は行いません。

```
FBFR *src;
char *dest;
. . .
if(Fmove(dest,src) < 0)
        F_error("pgm_name");
```

宛先バッファは、フィールド化バッファ (Falloc で割り当てられたバッファ) でなくてもかまいませんが、short 型境界に調整されている必要があります (FML32 では 4 バイト)。したがって、Fmove は、フィールド化バッファをフィールド化バッファ以外のバッファにコピーする必要がある時に、Fcpy の代替関数として機能します。ただし Fmove は、宛先バッファに十分な領域があるかどうかの確認は行いません。

値が FLD_PTR 型の場合、Fmove32 はバッファ・ポインタを転送します。アプリケーション・プログラマは、対応するポインタの移動に応じてバッファの再割り当て、および解放を管理しなければなりません。FLD_PTR フィールドが指すバッファは、tpalloc(3c) を呼び出して割り当てます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fmove、Fmove32(3fml)」を参照してください。

Fcpy

Fcpy は、あるフィールド化バッファを別のフィールド化バッファで上書きするために使用します。

```
int
Fcpy(FBFR *dest, FBFR *src)
```

以下はパラメータの説明です。

- *dest* は、宛先フィールド化バッファに対するポインタです。
- *src* は、ソース・フィールド化バッファに対するポインタです。

Fcpy は、上書きされたフィールド化バッファの全体の長さを保持するため、フィールド化バッファのサイズを拡大または縮小する場合に便利です。次の例を参照してください。

```
FBFR *src, *dest;
. . .
if(Fcpy(dest, src) < 0)
        F_error(“pgm_name”);
```

Fmove の *dest* は、初期化されていない領域を指す場合があります。しかし、Fcpy の *dest* には、初期化されたフィールド化バッファ (Falloc で割り当てられる) を指定する必要があります。さらに、Fcpy は、*dest* に指定されたバッファが十分な大きさであるかどうかも検証します。

注記　フィールド化バッファのサイズを、現在格納されているデータの領域より小さくすることはできません。

Fmove と同じく、Fcpy は、ソース・バッファを変更しません。

値が FLD_PTR 型の場合、Fcpy32 はバッファ・ポインタをコピーします。アプリケーション・プログラマは、関連するポインタがコピーされたときのバッファの再割り当てと解放を管理する必要があります。FLD_PTR フィールドが指すバッファは、tpalloc(3c) を呼び出して割り当てます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fcpy、Fcpy32(3fml)」を参照してください。

フィールドへのアクセスおよびフィールドの変更を行う関数

この節では、変換処理を行わずにフィールド型を使用して、フィールド化バッファにアクセスしたり、更新する方法を説明します。フィールド化バッファから、またはフィールド化バッファへのデータ転送時に、データ型を変換する関数の一覧については、5-54 ページの「変換を行う関数」を参照してください。

Fadd

Fadd は、フィールド化バッファに新しいフィールド値を追加します。

```
int
Fadd(FBFR *fbfr, FLDID fieldid, char *value, FLDLEN len)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *value* は、新しい値に対するポインタです。上の例は char* 型を示していますが、実際に使用するときは、追加する値と同じ型を指定します。後の例を参照してください。
- FLD_CARRAY 型または FLD_MBSTRING 型の場合、*len* は値の長さです。

バッファ内にフィールド・オカレンスがない場合は、フィールドが追加されます。1 つまたは複数のフィールド・オカレンスが既に存在する場合は、新しいフィールド・オカレンスとして値が追加され、現在指定されている最も大きい番号より 1 つ大きい番号が割り当てられます。特定のオカレンスを追加するには、Fchg を使用します。

Fadd には、フィールド値に対するポインタを指定し、フィールド値自体は指定しません。これは、フィールド値を取得したり、フィールド値を返すその他の関数でも同じです。

フィールド長が固定されているフィールド型 (short 型、long 型、char 型、float 型、double 型)、またはフィールド長を指定できるフィールド型 (string 型) の場合、フィールド長を指定する必要ありません。フィールド長は無視されます。フィールド型が文字配列 (FLD_CARRAY または FLD_MBSTRING) の場合は、フィールド長を指定する必要があります。フィールド長は FLDLEN 型で定義されます。たとえば、次のコードは、目的のフィールドのフィールド識別子を取得し、フィールド値をバッファに追加します。

```
FLDID fieldid, Fldid;
FBFR *fbfr;
. . .
fieldid = Fldid("fieldname");
```

```
if(Fadd(fbfr, fieldid, "new value", (FLDLEN)9) < 0)
          F_error("pgm_name");
```

デフォルトでは、文字配列型がネイティブなフィールド型と見なされます。したがって、関数には値の長さを渡す必要があります。追加する値が文字列配列以外の場合、value の型は、ポインタが指す値の型に応じて変わります。たとえば、次のコードは、long 型のフィールド値を追加します。

```
long lval;
. . .
lval = 123456789;
if(Fadd(fbfr, fieldid, &lval, (FLDLEN)0) < 0)
          F_error("pgm_name");
```

文字配列フィールドの場合、NULL フィールドは、長さ 0 で表します。文字列フィールドでは、フィールド値の一部として NULL 終了バイトが格納されるので、NULL 文字列を格納できます。つまり、NULL 終了バイトだけで構成される文字列は、長さ 1 であると見なされます。これ以外の型 (固定長の型) では、アプリケーション・プログラムで NULL と解釈される特殊な値を使用できますが、値のサイズは、実際に渡される値とは関係なく、フィールド型で指定されます (たとえば、long 型の場合は長さ 4)。NULL 値のアドレスを渡すとエラー (FEINVAL) が発生します。

ポインタ・フィールドの場合、Fadd32 はポインタ値を格納します。FLD_PTR フィールドが指すバッファは、tpalloc(3c) を呼び出して割り当てます。埋め込み型の FML32 バッファの場合、Fadd32 はインデックスを除くすべての FLD_FML32 フィールド値を格納します。

埋め込み型の VIEW32 バッファの場合、Fadd32 は FVIEWFLD 型の構造体に対するポインタを格納します。FVIEWFLD 型の構造体には、vflags (現在未使用で 0 に設定されているフラグ・フィールド)、vname (VIEW 名を含む文字配列)、および data (C 構造体として格納される VIEW データに対するポインタ) が含まれています。アプリケーションは、Fadd32 に vname と data を提供します。FVIEWFLD 構造体は、次のとおりです。

```
typedef struct {
     TM32U vflags;                /* フラグ - 現在未使用 */
     char vname[FVIEWNAMESIZE+1]; /* VIEW の名前 */
     char *data;                  /* VIEW 構造体に対するポインタ */
} FVIEWFLD;
```

詳細については、『BEA Tuxedo FML リファレンス』の「Fadd、Fadd32(3fml)」を参照してください。

Fappend

Fappend は、フィールド化バッファに新しいフィールド値を付加します。

```
int
Fappend(FBFR *fbfr, FLDID fieldid, char *value, FLDLEN len)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *value* は、新しい値に対するポインタです。上の例は char* 型を示していますが、実際に使用するときは、付加する値と同じ型を指定します。後の例を参照してください。
- FLD_CARRAY 型または FLD_MBSTRING 型の場合、*len* は値の長さです。

Fappend は、*value* で定義された値を指定した、新しい *fieldid* のフィールド・オカレンスを付加し、バッファを付加モードにします。付加モードでは、同じフィールド・セットを含む多数の行で構成された、サイズの大きいバッファが最適化されます。

バッファを付加モードにすると、バッファに対する操作が制限されます。付加モードで呼び出せる FML ルーチンは、Fappend、Findex、Funindex、Ffree、Fused、Funused および Fsizeof のみです。Findex または Funindex を呼び出すと、付加モードは終了します。

次の例では、Fappend を使用して、各行に 5 つのフィールドがある 500 行のバッファを作成します。

```
for (i=0; i 500 ;i++) {
   if ((Fappend(fbfr, LONGFLD1, &lval1[i], (FLDLEN)0) < 0) ||
      (Fappend(fbfr, LONGFLD2, &lval2[i], (FLDLEN)0) < 0) ||
      (Fappend(fbfr, STRFLD1, &str1[i], (FLDLEN)0) < 0) ||
      (Fappend(fbfr, STRFLD2, &str2[i], (FLDLEN)0) < 0) ||
      (Fappend(fbfr, LONGFLD3, &lval3[i], (FLDLEN)0) < 0)) {
      F_error("pgm_name");
```

```
      break;
    }
}
Findex(fbfr, 0);
```

Fappend には、フィールド値に対するポインタを指定し、フィールド値自体は指定しません。これは、フィールド値を取得したり、フィールド値を返すその他の関数でも同じです。

フィールド長が固定されているフィールド型 (short 型、long 型、char 型、float 型、double 型)、またはフィールド長を指定できるフィールド型 (string 型) の場合、フィールド長を指定する必要ありません。フィールド長は無視されます。フィールド型が文字配列 (FLD_CARRAY または FLD_MBSTRING) の場合は、フィールド長を指定する必要があります。フィールド長は FLDLEN 型で定義されます。

デフォルトでは、文字配列型がネイティブなフィールド型と見なされます。したがって、関数には値の長さを渡す必要があります。付加する値が文字列配列以外の場合、value の型は、ポインタが指す値の型に応じて変わります。

文字配列フィールドの場合、NULL フィールドは、長さ 0 で表します。文字列フィールドでは、フィールド値の一部として NULL 終了バイトが格納されるので、NULL 文字列を格納できます。つまり、NULL 終了バイトだけで構成される文字列は、長さ 1 であると見なされます。これ以外の型 (固定長の型) では、アプリケーション・プログラムで NULL と解釈される特殊な値を使用できますが、値のサイズは、実際に渡される値とは関係なく、フィールド型で指定されます (たとえば、long 型の場合は長さ 4)。NULL 値のアドレスを渡すとエラー (FEINVAL) が発生します。

詳細については、『BEA Tuxedo FML リファレンス』の「Fappend、Fappend32(3fml)」を参照してください。

Fchg

Fchg は、バッファ内のフィールド値を変更します。

```
int
Fchg(FBFR *fbfr, FLDID fieldid, FLDOCC oc, char *value, FLDLEN len)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、フィールドのオカレンス番号です。
- *value* は、新しい値に対するポインタです。上の例は char * 型を示していますが、実際に使用するときは、追加する値と同じ型を指定します。5-22 ページの「Fadd」を参照してください。
- FLD_CARRAY 型または FLD_MBSTRING 型の場合、*len* は値の長さです。

たとえば、次のコードは、carray 型のフィールドを value に格納された新しい値に変更します。

```
FBFR *fbfr;
FLDID fieldid;
FLDOCC oc;
FLDLEN len;
char value[50];
. . .
strcpy(value, "new value");
flen = strlen(value);
if(Fchg(fbfr, fieldid, oc, value, len) < 0)
       F_error("pgm_name");
```

oc が -1 の場合、フィールド値が新しいオカレンスとしてバッファに追加されます。oc が 0 以上であり、フィールドが見つかった場合、フィールド値は指定された新しい値に変更されます。oc が 0 以上であり、フィールドがない場合は、指定されたオカレンスとして値を追加できるまで、NULL オカレンスがバッファに追加されます。たとえば、バッファ上に存在しないフィールドのフィールド・オカレンス 3 を変更しようとすると、3 つの NULL オカレンス (オカレンス 0、1 および 2) が追加され、続いて、フィールド値が指定されたオカレンス 3 が追加されます。NULL 値については、文字列型と文字型の値の場合は NULL 文字列「\0」(長さ 1 バイト)、long 型と short 型のフィールドの場合は 0、float 型と double 型の値の場合は 0.0、文字配列の場合は 0 長の文字列が使用されます。

新しい値または変更された値は、value に格納されます。文字配列 (FLD_CARRAY または FLD_MBSTRING) の場合、長さは len で指定されます。その他のフィールド型では、len は無視されます。値のポインタが NULL であ

り、フィールドが見つかった場合、そのフィールドは削除されます。削除対象のフィールド・オカレンスが見つからないと、エラー (FNOTPRES) と見なされます。

ポインタ・フィールドの場合、Fchg32 にポインタ値が格納されます。FLD_PTR フィールドが指すバッファは、tpalloc(3c) を呼び出して割り当てます。埋め込み型の FML32 バッファの場合、Fchg32 は、インデックスを除くすべての FLD_FML32 フィールドの値を格納します。

埋め込み型の VIEW32 バッファの場合、Fchg32 は FVIEWFLD 型の構造体に対するポインタを格納します。FVIEWFLD 型の構造体には、vflags (現在未使用で 0 に設定されているフラグ・フィールド)、vname (VIEW 名を含む文字配列)、および data (C 構造体として格納される VIEW データに対するポインタ) が含まれています。アプリケーションは、vname と data を Fchg32 に提供します。FVIEWFLD 構造体は、次のとおりです。

```
typedef struct {
     TM32U vflags;                 /* フラグ - 現在未使用 */
     char vname[FVIEWNAMESIZE+1]; /* VIEW の名前 */
     char *data;                   /* VIEW 構造体に対するポインタ */
} FVIEWFLD;
```

バッファには、変更または追加されたフィールド値を格納できる領域が必要です。空き領域不足の場合、エラー (FNOSPACE) が返されます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fchg、Fchg32(3fml)」を参照してください。

Fcmp

Fcmp は、2 つのフィールド化バッファのフィールド識別子とフィールド値を比較します。

```
int
Fcmp(FBFR *fbfr1, FBFR *fbfr2)
```

fbfr1 および *fbfr2* は、フィールド化バッファに対するポインタです。

この関数は、2 つのバッファが同一の場合は 0 を返し、以下の条件のいずれかが成立する場合は -1 を返します。

- *fbfr1* フィールドのフィールド識別子が、対応する *fbfr2* フィールドのフィールド識別子より小さい。
- *fbfr1* フィールドの値が、対応する *fbfr2* フィールドの値より小さい。
- *fbfr1* が *fbfr2* より短い。

ポインタと埋め込み型のバッファが同一かどうかは、以下の条件によって決まります。

- ポインタ・フィールドの場合、ポインタ値 (アドレス) が同じであれば 2 つのポインタ・フィールドは同一と見なされます。
- 埋め込み型の FML32 バッファの場合、すべてのフィールド・オカレンスと値が同じであれば、2 つのフィールドは同一と見なされます。
- 埋め込み型の VIEW32 バッファの場合、VIEW 名が同じであり、さらに、すべての構造体メンバのオカレンスと値が同じであれば、2 つのフィールドは同一と見なされます。

`Fcmp` は、上記のいずれかの条件の逆が true である場合に `1` を返します。たとえば、*fbfr2* フィールドのフィールド識別子が *fbfr1* フィールドの対応するフィールド識別子より小さい場合、`Fcmp` は `1` を返します。

詳細については、『BEA Tuxedo FML リファレンス』の「`Fcmp、Fcmp32(3fml)`」を参照してください。

Fdel

`Fdel` は、指定されたフィールド・オカレンスを削除します。

```
int
Fdel(FBFR *fbfr, FLDID fieldid, FLDOCC oc)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、オカレンス番号です。

たとえば、次のコードは、指定されたフィールド識別子が示すフィールドの最初のオカレンスを削除します。

```
FLDOCC occurrence;
. . .
occurrence=0;
if(Fdel(fbfr, fieldid, occurrence) < 0)
            F_error("pgm_name");
```

指定されたフィールドが存在しない場合は -1 が返され、Ferror が FNOTPRES に設定されます。

ポインタ・フィールドの場合、Fdel32 は、参照されるバッファを変更したり、ポインタを解放しないで、FLD_PTR フィールド・オカレンスを削除します。データ・バッファは、オペークなポインタとして扱われます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fdel、Fdel32(3fml)」を参照してください。

Fdelall

Fdelall は、指定されたフィールドのすべてのオカレンスをバッファから削除します。

```
int
Fdelall(FBFR *fbfr, FLDID fieldid)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。

次の例を参照してください。

```
if(Fdelall(fbfr, fieldid) < 0)
     F_error("pgm_name");        /* フィールドが存在しない */
```

フィールドが見つからないと、-1 が返され、Ferror が FNOTPRES に設定されます。

ポインタ・フィールドの場合、Fdelall32 は、参照されるバッファを変更したり、ポインタを解放しないで、FLD_PTR フィールド・オカレンスを削除します。データ・バッファは、オペークなポインタとして扱われます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fdelall、Fdelall32(3fml)」を参照してください。

Fdelete

Fdelete は、フィールド識別子の配列 (fieldid[]) にリストされているすべてのフィールドのすべてのオカレンスを削除します。

```
int
Fdelete(FBFR *fbfr, FLDID *fieldid)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、削除対象となるフィールド識別子のリストに対するポインタです。

更新は、フィールド化バッファに対して直接行われます。フィールド識別子の配列内のエントリを特定の順序で並べる必要はありませんが、配列の最後のエントリは、フィールド識別子 0 (BADFLDID) でなければなりません。次の例を参照してください。

```
#include "fldtbl.h"
FBFR *dest;
FLDID fieldid[20];
.. .
fieldid[0] = A;   /* フィールド A のためのフィールド識別子 */
fieldid[1] = D;   /* フィールド D のためのフィールド識別子 */
fieldid[2] = BADFLDID;   /* 標識値 */
if(Fdelete(dest, fieldid) < 0)
        F_error("pgm_name");
```

宛先バッファに、A、B、C、D という 4 つのフィールドがある場合、上記の例では、フィールド B およびフィールド C のオカレンスのみを含むバッファが生成されます。

Fdelete を使用すると、Fdelall を数回呼び出す場合より効率的にバッファから複数のフィールドを削除できます。

ポインタ・フィールドの場合、Fdelete は、参照されるバッファを変更したり、ポインタを解放しないで、FLD_PTR フィールド・オカレンスを削除します。データ・バッファは、オペークなポインタとして扱われます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fdelete、Fdelete32(3fml)」を参照してください。

Ffind

Ffind は、バッファ内の指定されたフィールド・オカレンスの値を検索します。

```
char *
Ffind(FBFR *fbfr, FLDID fieldid, FLDOCC oc, FLDLEN *len)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、オカレンス番号です。
- *len* は、検索対象の値の長さです。

上の例では、Ffind の戻り値として、文字ポインタのデータ型 (C の char*) が示されています。実際に返されるポインタの型は、ポインタが指す値の型と同じです。

次のコードは、この関数の使用方法を示します。

```
#include "fldtbl.h"
FBFR *fbfr;
FLDLEN len;
char* Ffind, *value;
. . .
if((value=Ffind(fbfr,ZIP,0, &len)) == NULL)
      F_error("pgm_name");
```

フィールドが見つかると、フィールドの長さが len 内に返され (len が NULL の場合は、長さは返されない)、フィールドの位置が関数の値として返されます。フィールドが見つからないと、NULL が返され、Ferror が FNOTPRES に設定されます。

Ffind は、フィールドに対して読み取り専用でアクセスするために使用します。Ffind からの戻り値でバッファを変更することはできません。フィールド値を変更できるのは、Fadd または Fchg 関数のみです。この関数は、埋め込み型のバッファにある指定されたフィールドのオカレンスは調べません。

Ffind からの戻り値は、バッファが変更されない限り有効です。この値は、short 型境界では確実に調整されますが、long 型または double 型の境界では、フィールド型が long 型または double 型でも、調整されない場合があります値の調整については、この章の後の説明を参照してください。変数を正しく境界に調整する必要があるプロセッサでは、正しく調整されていない値を参照すると、システム・エラーが発生します。次は、その例です。

```
long *l1,l2;
FLDLEN length;
char *Ffind;
. . .
if((l1=(long *)Ffind(fbfr, ZIP, 0, &length)) == NULL)
        F_error("pgm_name");
else
        l2 = *l1;
```

このコードは、次のように書き直さなければなりません。

```
if((l1==(long *)Ffind(fbfr, ZIP, 0, &length)) == NULL)
         F_error("pgm_name");
else
         memcpy(&l2,l1,sizeof(long));
```

詳細については、『BEA Tuxedo FML リファレンス』の「Ffind、Ffind32(3fml)」を参照してください。

Ffindlast

この関数は、フィールド化バッファ内のフィールドの最後のオカレンスを検索し、そのフィールドに対するポインタと、最後のフィールド・オカレンスのオカレンス番号と長さを返します。

```
char *
Ffindlast(FBFR *fbfr, FLDID fieldid, FLDOCC *oc, FLDLEN *len)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、検索対象の最後のフィールド・オカレンスのオカレンス番号に対するポインタです。
- *len* は、検索対象の値の長さに対するポインタです。

上の例では、Ffindlast の戻り値として、文字ポインタのデータ型 (C の char*) が示されています。実際に返されるポインタの型は、ポインタが指す値の型と同じです。

Ffindlast は、Ffind と同様に動作します。ただし、フィールド・オカレンスを指定する必要はなく、関数の戻り値として、最後のフィールド・オカレンスのオカレンス番号と値が返されます。関数の呼び出し時にオカレンスに NULL を指定すると、オカレンス番号は返されません。この関数は、埋め込み型のバッファにある指定されたフィールドのオカレンスは調べません。

Ffindlast が返す値は、バッファが変更されない限り有効です。

詳細については、『BEA Tuxedo FML リファレンス』の「Ffindlast、Ffindlast32(3fml)」を参照してください。

Ffindocc

Ffindocc は、バッファ内の指定したフィールドのオカレンスを調べ、ユーザ指定のフィールド値と一致する最初のフィールド・オカレンスのオカレンス番号を返します。

```
FLDOCC
Ffindocc(FBFR *fbfr, FLDID fieldid, char *value, FLDLEN len;)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *value* は、新しい値に対するポインタです。上の例は char * 型を示していますが、実際に使用するときは、追加する値と同じ型を指定します。5-22 ページの「Fadd」を参照してください。
- FLD_CARRAY 型または FLD_MBSTRING 型の場合、*len* は値の長さです。

たとえば、次のコードは、oc を、指定された郵便番号のオカレンスに設定します。

```
#include "fldtbl.h"
FBFR *fbfr;
FLDOCC oc;
long zipvalue;
. . .
zipvalue = 123456;
if((oc=Ffindocc(fbfr,ZIP,&zipvalue, 0)) < 0)
       F_error("pgm_name");
```

文字列フィールドでは、正規表現がサポートされています。たとえば、次のコードは、oc を「J」で始まる NAME のオカレンスに設定します。

```
#include "fldtbl.h"
FBFR *fbfr;
FLDOCC oc;
char *name;
. . .
name = "J.*"
if ((oc = Ffindocc(fbfr, NAME, name, 1)) < 0)
        F_error("pgm_name");
```

注記　ただし、文字列上でのパターン照合を可能にするには、Ffindocc の 4 番目の引数を 0 以外にする必要があります。この引数が 0 の場合、単純な文字列比較が実行されます。フィールド値が見つからない場合は、-1 が返されます。

上位互換性のため、接頭辞としてアクセント記号 (^) 、また接尾辞としてドル記号 ($) が正規表現に暗黙的に追加されます。したがって、前の例にある正規表現は、実際には「^(J.*)$」として解釈されます。正規表現は、フィールド内の文字列値全体と一致しなければなりません。

詳細については、『BEA Tuxedo FML リファレンス』の「Ffindocc、Ffindocc32(3fml)」を参照してください。

Fget

Fget は、値が変更されたときに、フィールド化バッファのフィールドを検索するために使用されます。

```
int
Fget(FBFR *fbfr, FLDID fieldid, FLDOCC oc, char *loc, FLDLEN *maxlen)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、オカレンス番号です。
- *loc* は、フィールド値のコピー先バッファに対するポインタです。
- *maxlen* は、関数が呼び出されたときにソース・バッファの長さを参照し、フィールドの長さを返すポインタです。

呼び出し側プログラムは、Fget にプライベート・バッファに対するポインタとプライベート・バッファの長さを提供します。maxlen を NULL として指定すると、宛先バッファはフィールド値を格納できるだけのサイズを持つと想定され、バッファの長さは返されません。

Fget は、目的のフィールドがバッファにない場合 (FNOTPRES) または宛先バッファが小さすぎる場合 (FNOSPACE) にはエラーを返します。たとえば、次のコードは、郵便番号が文字配列または文字列として格納されている場合は、その郵便番号を検索します。

```
FLDLEN len;
char value[100];
. . .
len=sizeof(value);
if(Fget(fbfr, ZIP, 0, value, &len) < 0)
          F_error("pgm_name");
```

郵便番号が long 型として格納されている場合は、次のコードで郵便番号を取得できます。

```
FLDLEN len;
long value;
. . .
```

```
len = sizeof(value);
if(Fget(fbfr, ZIP, 0, value, &len) < 0)
          F_error("pgm_name");
```

詳細については、『BEA Tuxedo FML リファレンス』の「Fget、Fget32(3fml)」を参照してください。

Fgetalloc

Fgetalloc は、Fget と同じく、バッファ・フィールドを検索し、そのコピーを作成します。ただし、フィールドの領域は、malloc(3) への呼び出しによって取得します。

```
char *
Fgetalloc(FBFR *fbfr, FLDID fieldid, FLDOCC oc, FLDLEN *extralen)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、オカレンス番号です。
- *extralen* は、取得する長さに対するポインタ (関数の呼び出し時)、または取得した実際の長さに対するポインタ (関数の復帰時) です。

上記の例では、Fgetalloc の戻り値として、文字ポインタのデータ型 (C の char*) が示されています。実際に返されるポインタの型は、ポインタが指す値の型と同じです。

Fgetalloc が成功すると、正しく境界に調整されたバッファ・フィールドのコピーに対する有効なポインタが返されます。失敗すると、NULL が返されます。malloc(3) が失敗すると、Fgetalloc からエラーが返され、Ferror が FMALLOC に設定されます。

Fgetalloc の最後のパラメータには、予備の領域を指定します。たとえば、空き領域が不足した場合に、フィールド化バッファに値を再度指定する代わりに、取得済みの領域を拡張する場合に取得する領域です。成功すると、割り当てられたバッファの長さが extralen に返されます。次の例を参照してください。

```
FLDLEN extralen;
FBFR *fieldbfr
char *Fgetalloc;
. . .
extralen = 0;
if (fieldbfr = (FBFR *)Fgetalloc(fbfr, ZIP, 0, &extralen) == NULL)
         F_error("pgm_name");
```

Fgetalloc で取得した領域を free で解放する処理は、呼び出し側プログラムの役割です。

詳細については、『BEA Tuxedo FML リファレンス』の「Fgetalloc、Fgetalloc32(3fml)」を参照してください。

Fgetlast

Fgetlast は、値の変更時にフィールド化バッファからフィールドの最後のオカレンスを検索するために使用します。

```
int
Fgetlast(FBFR *fbfr, FLDID fieldid, FLDOCC *oc, char *loc, FLDLEN *maxlen)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、最後のフィールド・オカレンスのオカレンス番号に対するポインタです。
- *loc* は、フィールド値のコピー先バッファに対するポインタです。
- *maxlen* は、関数が呼び出されたときにソース・バッファの長さを参照し、フィールドの長さを返すポインタです。

呼び出し側プログラムは、Fgetlast にプライベート・バッファに対するポインタとプライベート・バッファの長さを提供します。Fgetlast は、Fget と同様に動作します。ただし、フィールド・オカレンスを指定する必要はなく、関数の戻り値として、最後のフィールド・オカレンスのオカレンス番号と値が返されます。ただし、occ に NULL を指定して関数を呼び出すと、オカレンス番号は返されません。

詳細については、『BEA Tuxedo FML リファレンス』の「Fgetlast、Fgetlast32(3fml)」を参照してください。

Fnext

Fnext は、指定されたフィールド・オカレンスの次のバッファ・フィールドを検索します。

```
int
Fnext(FBFR *fbfr, FLDID *fieldid, FLDOCC *oc, char *value, FLDLEN *len)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子に対するポインタです。
- *oc* は、オカレンス番号に対するポインタです。
- *value* は、次のフィールドに含まれる値と同じ型のポインタです。
- *len* は、**value* の指す長さに対するポインタです。

バッファ内の最初のフィールドを取得するには、fieldid に FIRSTFLDID を代入します。フィールド識別子と最初のフィールド・オカレンスのオカレンス番号が対応するパラメータ内に返されます。フィールドが NULL でない場合は、フィールド値が value ポインタでアドレス指定されたメモリ位置にコピーされます。

len パラメータを使用すると、フィールド値を格納できるだけの領域が value に割り当てられているかどうかを判別できます。十分な領域が割り当てられていない場合は、Ferror が FNOSPACE に設定されます。値の長さが

len パラメータ内に返されます。ただし、フィールドの値が NULL でない場合、len パラメータは、value に現在割り当てられている領域の長さも含んでいると見なします。

取り出されるフィールドが埋め込み型の VIEW32 バッファのときは、*value* パラメータは FVIEWFLD 構造体を指します。Fnext 関数は、構造体の *vname* フィールドと *data* フィールドを設定します。FVIEWFLD 構造体は、次のとおりです。

```
typedef struct {
     TM32U vflags;                 /* フラグ - 現在未使用 */
     char vname[FVIEWNAMESIZE+1]; /* VIEW の名前 */
     char *data;                   /* VIEW 構造体に対するポインタ */
} FVIEWFLD;
```

フィールド値が NULL の場合は、value パラメータと length パラメータは変更されません。

フィールドがそれ以上見つからない場合は、Fnext は 0 を返し (バッファの終わり)、fieldid、occurrence、value は変更されません。

value パラメータが NULL でない場合は、length パラメータも NULL でないと想定されます。

以下の例は、バッファ内のすべてのフィールド・オカレンスを読み取ります。

```
FLDID fieldid;
FLDOCC occurrence;
char *value[100];
FLDLEN len;
.. .
for(fieldid=FIRSTFLDID,len=sizeof(value);
     Fnext(fbfr,&fieldid,&occurrence,value,&len) > 0;
     len=sizeof(value)) {
   /* 各フィールド・オカレンス用のコード */
}
```

詳細については、『BEA Tuxedo FML リファレンス』の「Fnext、Fnext32(3fml)」を参照してください。

Fnum

Fnum は、指定されたバッファに含まれているフィールド数を返します。エラーの場合は -1 を返します。

```
FLDOCC
Fnum(FBFR *fbfr)
```

fbfr は、フィールド化バッファに対するポインタです。たとえば、次のコードは、指定されたバッファ内のフィールド数を出力します。

```
if((cnt=Fnum(fbfr)) < 0)
  F_error("pgm_name");
else
  fprintf(stdout,"%d fields in buffer\n",cnt);
```

FLD_FML32 および FLD_VIEW32 フィールドは、格納するフィールド数に関係なく、単一フィールドとしてカウントされます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fnum、Fnum32(3fml)」を参照してください。

Foccur

Foccur は、バッファ内の指定されたフィールドのオカレンス数を返します。

```
FLDOCC
Foccur(FBFR *fbfr, FLDID fieldid)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。

埋め込み型の FML32 バッファ内でのフィールドのオカレンスはカウントされません。

フィールド・オカレンスがバッファ内にない場合は 0 が返され、エラーの場合は -1 が返されます。たとえば、次のコードは、指定されたバッファ内のフィールド ZIP のオカレンス数を出力します。

```
FLDOCC cnt;
. . .
if((cnt=Foccur(fbfr,ZIP)) < 0)
 F_error("pgm_name");
else
 fprintf(stdout,"Field ZIP occurs %d times in buffer\n",cnt);
```

詳細については、『BEA Tuxedo FML リファレンス』の「Foccur、Foccur32(3fml)」を参照してください。

Fpres

Fpres、指定されたフィールド・オカレンスが存在する場合は true (1) を返し、存在しない場合は false (0) を返します。

```
int
Fpres(FBFR *fbfr, FLDID fieldid, FLDOCC oc)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、オカレンス番号です。

たとえば、次のコードは、fbfr が指すフィールド化バッファ内にフィールド ZIP が存在する場合、true を返します。

```
Fpres(fbfr,ZIP,0)
```

Fpres は、埋め込み型のバッファにある指定されたフィールドのオカレンスは確認しません。

詳細については、『BEA Tuxedo FML リファレンス』の「Fpres、Fpres32(3fml)」を参照してください。

Fvals および Fvall

Fvals は、string 値の場合は Ffind と同じように動作しますが、値に対するポインタを必ず返します。Fvall は、long 型および short 型の値の場合は、Ffind と同じように動作しますが、実際のフィールド値を、値に対するポインタの代わりに long 型で返します。

```
char*
Fvals(FBFR *fbfr,FLDID fieldid,FLDOCC oc)
```

```
char*
Fvall(FBFR *fbfr,FLDID fieldid,FLDOCC oc)
```

2 つの関数に共通するパラメータは、以下のとおりです。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子です。
- *oc* は、オカレンス番号です。

Fvals の場合、指定されたフィールド・オカレンスが見つからないと、NULL 文字列 \0 を返します。この関数は、戻り値をチェックせずにフィールドの値を別の関数に渡すのに役立ちます。ただし、この関数は、string 型のフィールドの場合のみ有効であり、ほかのフィールド型の場合は、自動的に NULL 文字列を返します (つまり、変換は行われません)。

Fvall の場合、指定されたフィールド・オカレンスが見つからないと、0 を返します。この関数は、戻り値をチェックせずにフィールドの値を別の関数に渡すのに役立ちます。ただし、この関数は、long 型と short 型のフィールドの場合のみ有効であり、ほかのフィールド型の場合は、自動的に 0 を返します (つまり、変換は行われません)。

詳細については、『BEA Tuxedo FML リファレンス』の「Fvals、Fvals32(3fml)」および「Fvall、Fvall32(3fml)」を参照してください。

バッファを更新する関数

この節では、バッファ内の個々のフィールドではなく、フィールド化バッファ全体にアクセスし、内容を更新する関数を説明します。これらの関数では、最大 3 つのパラメータしか使用されません。

- *dest* は、宛先フィールド化バッファに対するポインタです。
- *src* は、ソース・フィールド化バッファに対するポインタです。
- *fieldid* は、フィールド識別子またはフィールド識別子の配列です。

Fconcat

Fconcat は、ソース・バッファのフィールドを、既存の宛先バッファのフィールドに追加します。

```
int
Fconcat(FBFR *dest, FBFR *src)
```

宛先バッファ内のオカレンスは保持されます。つまり、このオカレンスは変更されません。新しいオカレンス、つまり、ソース・バッファから取得されたオカレンスには、宛先フィールドの既存のオカレンス番号より大きい番号が設定され、宛先バッファに追加されます。フィールドは、フィールド識別子の順序で保持されます。

次の例を参照してください。

```
FBFR *src, *dest;
. . .
if(Fconcat(dest,src) < 0)
       F_error("pgm_name");
```

dest に 2 つのフィールド (A、B) と 2 つのオカレンスを持つフィールド C があり、src に 3 つのフィールド (A、C、D) があるとします。この結果、dest には、2 つのオカレンスを持つフィールド A (宛先のフィールド A とソースのフィールド A)、フィールド B、3 つのオカレンスを持つフィールド C (dest の 2 つのオカレンスと src のオカレンス)、およびフィールド D が設定されます。

新しいフィールドを格納できるだけの領域がない場合 (FNOSPACE が返された場合)、この処理は失敗し、宛先バッファは変更されません。

詳細については、『BEA Tuxedo FML リファレンス』の「Fconcat、Fconcat32(3fml)」を参照してください。

Fjoin

Fjoin は、フィールド識別子とオカレンスの組み合わせに基づき、2 つのフィールド化バッファを結合します。

```
int
Fjoin(FBFR *dest, FBFR *src)
```

フィールド識別子とオカレンスの組み合わせが一致するフィールド間では、宛先バッファの値がソース・バッファの値で更新されます。ソース・バッファ内に、対応するフィールド識別子とオカレンスの組み合わせがない場合、宛先バッファのフィールドは削除されます。宛先バッファ内に、対応するフィールド識別子とオカレンスの組み合わせがない場合、ソース・バッファ内のフィールドは、宛先バッファに追加されません。次の例を参照してください。

```
if(Fjoin(dest,src) < 0)
     F_error("pgm_name");
```

Fconcat の例で使用した入力バッファを使用すると、ソース・フィールド値 A とソース・フィールド値 C を持つ宛先バッファが生成されます。新しい値が古い値より大きい場合は、この関数は領域の不足のために失敗することがあります (FNOSPACE)。その場合、宛先バッファは変更されていると予測されます。ただし、この事態が発生した場合、Frealloc 関数と Fjoin 関数を繰り返し使用すると、宛先バッファを再割り当てできます (宛先バッファが部分的に更新されてしまった場合でも、これらの関数を繰り返し使用すると、正しい結果が得られます)。

バッファの結合によってポインタ・フィールド (FLD_PTR) が削除されると、ポインタが参照するメモリ領域は、変更も解放もされません。

詳細については、『BEA Tuxedo FML リファレンス』の「Fjoin、Fjoin32(3fml)」を参照してください。

Fojoin

Fojoin は、Fjoin と似ていますが、ソース・バッファ内に対応するフィールド識別子とオカレンスの組み合わせがない宛先バッファのフィールドの削除は行いません。

```
int
Fojoin(FBFR *dest, FBFR *src)
```

宛先バッファ内に対応するフィールド識別子とオカレンスの組み合わせがないソース・バッファ内のフィールドは、宛先バッファに追加されません。次の例を参照してください。

```
if(Fojoin(dest,src) < 0)
      F_error("pgm_name");
```

Fjoin の例で使用した入力バッファを使用すると、この呼び出しの結果の dest には、ソース・フィールド値 A、宛先フィールド値 B、ソース・フィールド値 C が含まれます。Fjoin の場合のように、Fojoin 関数は、領域の不足のために失敗する場合があり (FNOSPACE)、その場合は、領域をさらに割り当てた後で関数を再発行すると、処理を完了できます。

バッファの結合によってポインタ・フィールド (FLD_PTR) が削除されると、ポインタが参照するメモリ領域は、変更も解放もされません。

詳細については、『BEA Tuxedo FML リファレンス』の「Fojoin、Fojoin32(3fml)」を参照してください。

Fproj

Fproj は、目的のフィールドのみが保存されるように、バッファの適切な箇所を更新します (つまり、結果として、指定されたフィールドのプロジェクションが行われます)。バッファの更新によってポインタ・フィールド (FLD_PTR) が削除されると、ポインタが参照するメモリ領域は、変更も解放もされません。

```
int
Fproj(FBFR *fbfr, FLDID *fieldid)
```

これらのフィールドは、この関数に渡されるフィールド識別子の配列で指定されます。更新は、フィールド化バッファ内で直接実行されます。次の例を参照してください。

```
#include "fldtbl.h"
FBFR *fbfr;
FLDID fieldid[20];
.. .
fieldid[0] = A;   /* フィールド A のためのフィールド識別子 */
fieldid[1] = D;   /* フィールド D のためのフィールド識別子 */
fieldid[2] = BADFLDID;   /* 標識値 */
if(Fproj(fbfr, fieldid) < 0)
      F_error("pgm_name");
```

バッファに A、B、C の各フィールドがある場合は、上記の例の結果として、フィールド A とフィールド D のオカレンスのみを含むバッファが生成されます。ただし、フィールド識別子の配列内のエントリは、特定の順序で並べる必要はありませんが、フィールド識別子 0 (BADFLDID) がフィールド識別子の配列の最後の値でなければなりません。

詳細については、『BEA Tuxedo FML リファレンス』の「Fproj、Fproj32(3fml)」を参照してください。

Fprojcpy

Fprojcpy は Fproj と似ていますが、目的のフィールドは宛先バッファに配置されます。バッファの更新によってポインタ・フィールド (FLD_PTR) が削除されると、ポインタが参照するメモリ領域は、変更も解放もされません。

```
int
Fprojcpy(FBFR *dest, FBFR *src, FLDID *fieldid)
```

まず、宛先バッファ内のすべてのフィールドが削除され、ソース・バッファでのプロジェクションの結果が宛先バッファにコピーされます。上記の例を使って、次のコードではプロジェクションの結果が宛先バッファに格納されます。

```
if(Fprojcpy(dest, src, fieldid) < 0)
       F_error("pgm_name");
```

フィールド識別子の配列内のエントリは、再配置される場合があります。つまり、フィールド識別子の配列は、それらのエントリが番号順になっていないとソートされます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fprojcpy、Fprojcpy32(3fml)」を参照してください。

Fupdate

Fupdate は、ソース・バッファ内のフィールド値で宛先バッファを更新します。

```
int
Fupdate(FBFR *dest, FBFR *src)
```

フィールド識別子とオカレンスの組み合わせが一致するフィールドの場合、フィールド値は、ソース・バッファ内の値によって宛先バッファ内で更新されます (Fjoin と同じ)。ソース・バッファに対応するフィールドがない宛先バッファのフィールドは、変更されません (Fojoin と同じ)。宛先バッファに対応するフィールドのないソース・バッファのフィールドは、宛先バッファに追加されます (Fconcat と同じ)。次の例を参照してください。

```
if(Fupdate(dest,src) < 0)
     F_error("pgm_name");
```

src バッファにフィールド A、C、D という 3 つのフィールドがあり、dest バッファにフィールド A、B という 2 つのフィールドと、フィールド C の 2 つのオカレンスがある場合、結果はソース・フィールド A、宛先フィールド B、ソース・フィールド C、2 つ目の宛先フィールド C、およびソース・フィールド D になります。

ポインタの場合、Fupdate32 はポインタ値を格納します。FLD_PTR フィールドが指すバッファは、tpalloc(3c) を呼び出して割り当てます。埋め込み型の FML32 バッファの場合、Fupdate32 は、インデックスを除くすべての FLD_FML32 フィールド値を格納します。

埋め込み型の VIEW32 バッファの場合、Fupdate32 は FVIEWFLD 型の構造体に対するポインタを格納します。FVIEWFLD 型の構造体には、vflags (現在未使用で 0 に設定されているフラグ・フィールド)、vname (VIEW 名を含

む文字配列)、および data (C 構造体として格納される VIEW データに対するポインタ) が含まれています。アプリケーションは、Fupdate32 に vname および data を提供します。FVIEWFLD 構造体は、次のとおりです。

```
typedef struct {
     TM32U vflags;                 /* フラグ - 現在未使用 */
     char vname[FVIEWNAMESIZE+1]; /* VIEW の名前 */
     char *data;                   /* VIEW 構造体に対するポインタ */
} FVIEWFLD;
```

詳細については、『BEA Tuxedo FML リファレンス』の「Fupdate、Fupdate32(3fml)」を参照してください。

VIEWS 関数

Fvftos

Fvftos は、指定された VIEW 記述を使用して、フィールド化バッファから C 構造体へデータを転送します。

```
int
Fvftos(FBFR *fbfr, char *cstruct, char *view)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *cstruct* は、構造体に対するポインタです。
- view は、VIEW 記述名の文字列に対するポインタです。

指定された VIEW が見つからない場合、Fvftos から -1 が返され、Ferror が FBADVIEW に設定されます。

フィールド化バッファから C 構造体へのデータ転送時には、以下の規則が適用されます。

- C 構造体のメンバにマッピングされていないフィールド化バッファのフィールドは無視されます。
- フィールド化バッファ内には存在しなくても、VIEW 記述には記述されており、構造体メンバにマッピングされているフィールドの場合は、対応する NULL 値がメンバ内にコピーされます。
- フィールド化バッファのフィールドが `string` 型または `carray` 型のデータを含む場合は、最大でマッピング先の構造体メンバのサイズまで文字がコピーされます (それ以上のマッピング元の値は切り捨てられます)。マッピング元の値がマッピング先の構造体メンバよりも短い場合は、メンバ値の残りに NULL 文字 (0) が埋め込まれます。string 型の値は、値を切り捨てても、必ず NULL 文字で終了します。
- バッファ内のフィールドのオカレンス数がマッピング先の構造体メンバの数と等しい場合は、フィールド化データは C 構造体にコピーされます。
- バッファ内のフィールドのオカレンス数がマッピング先の構造体メンバの数より多い場合は、フィールド化データは無視されます。
- バッファ内のフィールドのオカレンス数がマッピング先の構造体メンバの数より少ない場合は、余分なメンバには対応する NULL 値が割り当てられます。

たとえば、次のコードは、`string1` を `cust.action[0]` に格納し、`abc` を `cust.bug[0]` に格納します。`cust` 構造体内のほかのすべてのメンバには NULL 値を格納します。

```
#include <stdio.h>
#include "fml.h"
#include "custdb.flds.h"
#include "custdb.h"
struct custdb cust;
FBFR *fbfr;
. . .
fbfr = Falloc(800,1000);
Fvinit((char *)&cust,"custdb");    /* cust 構造体の初期化 */
str = "string1";
Fadd(fbfr,ACTION,str,(FLDLEN)8);
str = "abc";
Fadd(fbfr,BUG_CURS,str,(FLDLEN)4);
Fvftos(fbfr,(char *)&cust,"custdb");
. . .
```

VIEW 記述 custdb の定義については、6-1 ページの「VIEWS の使用例」を参照してください。

詳細については、『BEA Tuxedo FML リファレンス』の「Fvftos、Fvftos32(3fml)」を参照してください。

Fvstof

Fvstof は、指定された VIEW 記述を使用して、C 構造体からフィールド化バッファへデータを転送します。

```
int
Fvstof(FBFR *fbfr, char *cstruct, int mode, char *view)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *cstruct* は、構造体に対するポインタです。
- *mode* は、FUPDATE、FJOIN、FOJOIN、FCONCAT のいずれかです。
- *view* は、VIEW 記述名の文字列に対するポインタです。

転送プロセスは、mode パラメータに対応する FML 関数、Fupdate、Fjoin、Fojoin、または Fconcat の項目で説明されている規則に従います。

指定された VIEW が見つからない場合、Fvstof から -1 が返され、Ferror が FBADVIEW に設定されます。

注記　NULL 値は、構造体メンバからフィールド化バッファに転送されません。つまり、構造体からフィールドへの転送時には、構造体メンバに対して定義されたデフォルト設定またはユーザ指定の NULL 値が構造体メンバに含まれていると、そのメンバは無視されます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fvftos、Fvftos32(3fml)」を参照してください。

Fvnull

Fvnull は、C 構造体内のオカレンスにそのフィールド用の NULL 値が含まれているかどうかを判別します。

```
int
Fvnull(char *cstruct, char *cname, FLDOCC oc, char *view)
```

以下はパラメータの説明です。

- *cstruct* は、構造体に対するポインタです。
- *cname* は、構造体メンバの名前に対するポインタです。
- *oc* は、特定要素に対するインデックスです。
- *view* は、VIEW 記述名の文字列に対するポインタです。

以下は、Fvnull の戻り値です。

- 1 (オカレンスが NULL の場合)
- 0 (オカレンスが NULL でない場合)
- -1 (エラーが発生した場合)

詳細については、『BEA Tuxedo FML リファレンス』の「Fvnull、Fvnull32(3fml)」を参照してください。

Fvsinit

Fvsinit は、適切な NULL 値で C 構造体内のすべての要素を初期化します。

```
int
Fvsinit(char *cstruct, char *view)
```

以下はパラメータの説明です。

- *cstruct* は、構造体に対するポインタです。
- *view* は、VIEW 記述名の文字列に対するポインタです。

詳細については、『BEA Tuxedo FML リファレンス』の「Fvsinit、Fvsinit32(3fml)」を参照してください。

Fvopt

Fvopt は、実行時にフラグ・オプションを変更します。

```
int
Fvopt(char *cname, int option, char *view)
```

以下はパラメータの説明です。

- *cname* は、構造体メンバの名前です。
- *option* は、以下のオプションのいずれかです。
- *view* は、VIEW 記述名の文字列に対するポインタです。

以下は、*option* パラメータに指定できる値の一覧です。

F_FTOS
: フィールド化バッファから C 構造体への一方向マッピングを指定します。VIEW 記述の S オプションと同じように機能します。

F_STOF
: C 構造体からフィールド化バッファへの一方向マッピングを指定します。VIEW 記述の F オプションと同じように機能します。

F_BOTH
: C 構造体とフィールド化バッファの間の双方向マッピングを指定します。

F_OFF
: 指定されたメンバのマッピングを無効にします。VIEW 記述の N オプションと同じように機能します。

VIEW 記述への変更は永久に保持されるわけではありません。変更内容の有効期間は、別の VIEW 記述に対してアクセスが行われるときまでです。

詳細については、『BEA Tuxedo FML リファレンス』の「Fvopt、Fvopt32(3fml)」を参照してください。

Fvselinit

Fvselinit は、C 構造体の個々のメンバを、適切な NULL 値に初期化します。この関数は、viewfile 内で C フラグが使用されていると、要素の ACM を 0 に設定します。viewfile 内で L フラグが使用されていると、要素の ALM を対応する NULL 値の長さに設定します。

```
int
Fvselinit(char *cstruct, char *cname, char *view)
```

以下はパラメータの説明です。

- *cstruct* は、構造体に対するポインタです。
- *cname* は、構造体メンバの名前に対するポインタです。
- *view* は、VIEW 記述名の文字列に対するポインタです。

詳細については、『BEA Tuxedo FML リファレンス』の「Fvselinit、Fvselinit32(3fml)」を参照してください。

変換を行う関数

FML には、フィールド化バッファの読み取りまたは書き込み時にデータを変換する一連のルーチンが用意されています。

一般に、これらのルーチンは、対応する非変換関数と同じように動作します。ただし、バッファへの書き込み時にはユーザ型からネイティブ型へ変換を行い、バッファからの読み取り時にはネイティブ型からユーザ型への変換を行います。

フィールドのネイティブ型は、そのフィールド・テーブル・エントリ内でそのフィールドに対して指定され、そのフィールド識別子内で符号化されたデータ型です。ただし、上記の規則に対する唯一の例外として CFfindocc があります。この関数は、読み取り操作を行いますが、ユーザ型からネイティブ型に変換を行ってから Ffindocc を呼び出します。これらの関数の名前は、接頭辞「C」が付いた、対応する非変換 FML 関数と同じです。

変換を行う関数では、ポインタ (FLD_PTR)、埋め込み型の FML32 バッファ (FLD_FML32)、および埋め込み型の VIEW32 バッファ (FLD_VIEW32) はサポートされていません。FML32 変換関数の実行中にこれらのフィールド型が使用されていた場合、Ferror が FEBADOP に設定されます。

CFadd

CFadd は、バッファにユーザ指定の項目を追加して、バッファ内に新しいフィールド・オカレンスを生成します。

```
int
CFadd(FBFR *fbfr, FLDID fieldid, char *value, FLDLEN len, int type)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、追加するフィールドのフィールド識別子です。
- *value* は、追加する値に対するポインタです。
- FLD_CARRAY 型の場合、*len* は値の長さです。
- *type* は、値の型です。

フィールドを追加する前に、データ項目は、ユーザ指定の型からフィールドをフィールド化バッファに格納される型としてフィールド・テーブル内で指定された型に変換されます。ソースが FLD_CARRAY 型 (文字配列) である場合は、len 引数を配列の長さに設定する必要があります。次の例を参照してください。

```
if(CFadd(fbfr,ZIP,"12345",(FLDLEN)0,FLD_STRING) < 0)
      F_error("pgm_name");
```

上記の例では、ZIP (郵便番号) フィールドが long 型整数としてフィールド化バッファ内に格納されている場合は、「12345」が long 型整数の表現に変換されてから、その表現が fbfr の指すフィールド化バッファに追加されます (ただし、フィールド値の長さは、関数が決定できるので 0 に指定されて

います。この長さは、FLD_CARRAY 型の場合のみ必要です)。以下の例は、同じ値をフィールド化バッファに格納しますが、その値を string 型としてではなく long 型として表現して格納します。

```
long zipval;
. . .
zipval = 12345;
if(CFadd(fbfr,ZIP,&zipval,(FLDLEN)0,FLD_LONG) < 0)
      F_error("pgm_name");
```

ただし、C では構造体 &12345L を使用できないので、まず、値を変数に格納する必要があります。CFadd は、成功すると 1 を返し、エラーになると -1 を返します (Ferror は、適宜に設定されます)。

詳細については、『BEA Tuxedo FML リファレンス』の「CFadd、CFadd32(3fml)」を参照してください。

CFchg

CFchg は、CFadd と同じように動作しますが、指定された値の変換後、フィールド値を変更します。

```
int
CFchg(FBFR *fbfr, FLDID fieldid, FLDOCC oc, char *value, FLDLEN len, int type)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、変更するフィールドのフィールド識別子です。
- *oc* は、変更するフィールドのオカレンス番号です。
- *value* は、追加する値に対するポインタです。
- FLD_CARRAY 型の場合、*len* は値の長さです。
- *type* は、値の型です。

たとえば、次のコードは、フィールド ZIP の最初のオカレンス (オカレンス 0) を指定された値に変更し、必要に応じて変換を行います。

```
FLDOCC occurrence;
long zipval;
. . .
zipval = 12345;
occurrence = 0;
if(CFchg(fbfr,ZIP,occurrence,&zipval,(FLDLEN)0,FLD_LONG) < 0)
       F_error("pgm_name");
```

指定されたオカレンスが見つからないと、指定されたオカレンスとして値を追加できるまで、NULL オカレンスがバッファに追加されます。

詳細については、『BEA Tuxedo FML リファレンス』の「CFchg、CFchg32(3fml)」を参照してください。

CFget

CFget は、Fget に似た変換関数です。ただし、CFget は変換した値をユーザ指定のバッファにコピーします。

```
int
CFget(FBFR *fbfr, FLDID fieldid, FLDOCC oc, char *buf, FLDLEN *len, int type)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、検索するフィールドのフィールド識別子です。
- *oc* は、フィールドのオカレンス番号です。
- *buf* は、変換後のコピー先バッファに対するポインタです。
- FLD_CARRAY 型の場合、*len* は値の長さです。
- *type* は、値の型です。

前の例を使って、次のコードは、バッファに格納されたばかりの値 (どのような形式の値でも可) にアクセスして、その値を元の long 型整数に戻します。

```
FLDLEN len;
. . .
len=sizeof(zipval);
```

```
if(CFget(fbfr,ZIP,occurrence,&zipval,&len,FLD_LONG) < 0)
        F_error("pgm_name");
```

長さのポインタが NULL の場合は、検索および変換した値の長さは返されません。

詳細については、『BEA Tuxedo FML リファレンス』の「CFget、CFget32(3fml)」を参照してください。

CFgetalloc

CFgetalloc は、Fgetalloc と同じように動作します。ただし、戻り値 (変換後の値) に対して malloc を使用して割り当てた領域は、free を使用して解放する必要があります。

```
char *
CFgetalloc(FBFR *fbfr, FLDID fieldid, FLDOCC oc, int type, FLDLEN *extralen)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、変換するフィールドのフィールド識別子です。
- *oc* は、フィールドのオカレンス番号です。
- *type* は、値の変換後の型です。
- *extralen* は、関数の呼び出し時には追加の割り当て量に対するポインタであり、関数の復帰時には割り当てられた総領域のサイズに対するポインタです。

上記の例では、CFgetalloc の戻り値として文字ポインタのデータ型 (C の char*) が示されています。実際に返されるポインタの型は、ポインタが指す値の型と同じです。

以下のコードを使用すると、既に格納されている値を自動的に割り当てられた領域に取り込むことができます。

```
char *value;
FLDLEN extra;
. . .
```

```
extra = 25;
if((value=CFgetalloc(fbfr,ZIP,0,FLD_LONG,&extra)) == NULL)
 F_error("pgm_name");
```

関数呼び出しに値 extra を指定した場合、関数は、取り出した値に十分な領域に加えて、さらに 25 バイトを割り当てます。割り当てられた領域の総量が、この変数に返されます。

詳細については、『BEA Tuxedo FML リファレンス』の「CFgetalloc、CFgetalloc32(3fml)」を参照してください。

CFfind

CFfind は、検索対象のフィールドの値を変換し、その値に対するポインタを返します。

```
char *
CFfind(FBFR *fbfr, FLDID fieldid, FLDOCC oc, FLDLEN len, int type)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、検索するフィールドのフィールド識別子です。
- *oc* は、フィールドのオカレンス番号です。
- *len* は、変換後の値の長さです。
- *type* は、値の変換後の型です。

上記の例では、CFfind の戻り値として、文字ポインタのデータ型 (C の char*) が示されています。実際に返されるポインタの型は、ポインタが指す値の型と同じです。

この関数が返すポインタは、Ffind の場合と同じく、読み取り専用と見なされます。たとえば、次のコードは、ZIP フィールドの最初のオカレンスの値を含む long 型に対するポインタを返します。

```
char *CFfind;
FLDLEN len;
long *value;
. . .
if((value=(long *)CFfind(fbfr,ZIP,occurrence,&len,FLD_LONG))== NULL)
   F_error("pgm_name");
```

長さに対するポインタが NULL の場合、検出された値の長さは返されません。この関数の戻り値は、Ffind の場合と異なり、対応するユーザ指定の型の境界に正しく調整されます。

注記　CFind が返すポインタは、次のバッファ操作 (破壊的でない操作も含む) が実行されるまで有効です。これは、変換後の値が 1 つのプライベート・バッファに保存されているためです。一方、Ffind の戻り値の場合は、次にバッファが変更されるまで有効です。

詳細については、『BEA Tuxedo FML リファレンス』の「CFfind、CFfind32(3fml)」を参照してください。

CFfindocc

CFfindocc は、バッファの指定されたフィールドのオカレンスを調べ、フィールド識別子の型に変換されたユーザ指定のフィールド値と一致する最初のフィールド・オカレンスのオカレンス番号を返します。

```
FLDOCC
CFfindocc(FBFR *fbfr, FLDID fieldid, char *value, FLDLEN len, int type)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *fieldid* は、検索するフィールドのフィールド識別子です。
- *value* は、対応する未変換の値に対するポインタです。
- *len* は、対応する未変換の値の長さです。
- *type* は、未変換の値の型です。

たとえば、次のコードは、文字列を fieldid ZIP の型 (おそらく long 型) に変換し、oc を指定された郵便番号のオカレンスに設定します。

```
#include "fldtbl.h"
FBFR *fbfr;
FLDOCC oc;
char zipvalue[20];
. . .
strcpy(zipvalue,"123456");
if((oc=CFfindocc(fbfr,ZIP,zipvalue,0,FLD_STRING)) <  0)
        F_error("pgm_name");
```

フィールド値が見つからない場合は、-1 が返されます。

注記　CFfindocc は、ユーザ指定の値をネイティブ・フィールド型に変換してから、フィールド値を検証するため、正規表現は、ユーザ指定の型とネイティブ・フィールド型が両方とも FLD_STRING である場合のみ機能します。したがって、CFfindocc には、正規表現に関するユーティリティがありません。

詳細については、『BEA Tuxedo FML リファレンス』の「CFfindocc、CFfindocc32(3fml)」を参照してください。

文字列を変換する関数

ユーザ指定の型 FLD_STRING への変換および FLD_STRING からの変換を処理するために、以下の関数が提供されています。

- Fadds、Fadds32(3fml)
- Fchgs、Fchgs32(3fml)
- Ffinds、Ffinds32(3fml)
- Fgets、Fgets32(3fml)
- Fgetsa、Fgetsa32(3fml)

これらの関数は、対応する非文字列用の関数を呼び出し、FLD_STRING 型と len 0 を提供します。ただし、Ffinds が返すポインタの有効期間は、CFfind の場合と同じです。

これらの関数の説明については、『BEA Tuxedo FML リファレンス』を参照してください。

Ftypcvt

CFadd、CFchg、CFget、CFgetalloc、CFfind の各関数は、Ftypcvt を使用して、適切なデータ変換を行います。Ftypcvt32 関数は、フィールド型 FLD_PTR、FLD_FML32、および FLD_VIEW32 では失敗します。Ftypcvt を使用する場合の形式は、次のとおりです。この形式は、パラメータの順序に関する規則に準拠しません。

```
char *
Ftypcvt(FLDLEN *tolen, int totype, char *fromval, int fromtype, FLDLEN fromlen)
```

以下はパラメータの説明です。

- *tolen* は、変換後の値の長さに対するポインタです。
- *totype* は、変換先の型です。
- *fromval* は、変換前の値に対するポインタです。
- *fromtype* は、変換前の型です。
- *fromlen* は、変換前の型が FLD_CARRAY である場合の変換前の値の長さです。

Ftypcvt は、fromlen で指定された長さの fromtype 型の *fromval の値 (fromtype が FLD_CARRAY の場合。それ以外の場合、fromlen は fromtype から算出される。) を、totype 型の値に変換します。Ftypcvt が成功すると、変換後の値に対するポインタが返され、*tolen に変換後の長さが設定されます。失敗すると、Ftypcvt から NULL が返されます。CFchg 関数を使用した次の例を参照してください。

```
CFchg(fbfr,fieldid,oc,value,len,type)
FBFR *fbfr;             /* フィールド化バッファ */
FLDID fieldid;          /* 変更されるべきフィールド */
FLDOCC oc;              /* 変更されるべきフィールドのオカレンス */
char *value;            /* 新しい値に対するポインタ */
FLDLEN len;             /* 新しい値の長さ */
int type;               /* 新しい値のタイプ */
```

```
{
 char *convloc;          /* 変換された値に対するポインタ */
 FLDLEN convlen;         /* 変換された値の長さ */
 extern char *Ftypcvt;

         /* 値をフィールド化バッファ・タイプに変換 */
  if((convloc = Ftypcvt(&convlen,FLDTYPE(fieldid),value,type,len)) == NULL)
                return(-1);

  if(Fchg(fbfr,fieldid,oc,convloc,convlen) < 0)
                return(-1);
  return(1);
}
```

Ftypcvt を直接呼び出して、フィールド化バッファを変更せずにフィールド値を変換することができます。

詳細については、『BEA Tuxedo FML リファレンス』の「Ftypcvt、Ftypcvt32(3fml)」を参照してください。

変換の規則

以下は、変換時の規則の一覧です。oldval は、変換対象のデータ項目に対するポインタを表し、newval は変換後の値に対するポインタを表します。

- 変換前と変換後の型が同一である場合、*newval と *oldval は同一です。
- 変換前と変換後の型が数値型 (long 型、short 型、float 型、または double 型) である場合は、C の代入演算子で正しい型変換を行います。たとえば、次のコードによって short 型は float 型に変換されます。

  ```
  *((float *)newval) = *((short *) oldval)
  ```

- 数値型から文字列型へ変換する場合は、適切な sprintf を使用します。たとえば、次のコードによって short 型は string 型に変換されます。

  ```
  sprintf(newval,"%d",*((short *)oldval))
  ```

- 文字列型から数値型へ変換する場合は、適切な関数 (たとえば、atof、atol) を使用し、その結果を型変換のために代入します。たとえば、次のように指定します。

  ```
  *((float *)newval) = atof(oldval)
  ```

- char 型から任意の数値型へ変換するか、または数値型から char 型へ変換する場合、char 型は、「短い short 型」と見なされます。たとえば、次のコードによって char 型は float 型に変換されます。

  ```
  *((float *)newval) = *((char *)oldval)
  ```

  short 型から char 型へ変換するには、次の例に示す方法を使用します。

  ```
  *((char *)newval) = *((short *)oldval)
  ```

- char 型は、NULL 文字を追加することによって string 型に変換されます。この場合の char 型は、「短い short 型」ではありません。char 型が short 型である場合は、char 型を short 型に変換してから sprintf で short 型を string 型に変換します。同様に、string の最初の文字を代入すると、string 型は、char 型に変換されます。

- carray 型は、任意のシーケンスのバイトを格納するために使用されます。このため、carray 型は、任意のユーザ・データ型を符号化できます。ただし、carray 型の場合は、以下の規則に従います。

  - carray 型は、carray 型に NULL バイトを付加することによって string 型に変換されます。このため、string から後続の NULL のオーバーヘッドを差し引いたものを格納したい場合は、carray 型を使用できます (ただし、フィールドは、フィールド化バッファ内で short 型の範囲内に調整されるので、必ず領域を節約できるとは限りません)。string は、その終了 NULL バイトを削除することによって carray 型に変換されます。

  - carray 型を任意の数値型に変換する場合は、まず、string 型に変換し、その string 型を数値型に変換します。同様に、数値型を carray 型に変換する場合は、まず、数値型を string 型に変換してから、string 型を carray 型に変換します。

  - 配列の最初の文字を char に代入すると、carray 型は char 型に変換されます。同様に、char 型を carray 型に変換する場合は、文字を配列の最初のバイトとして代入し、配列の長さを 1 に設定します。

  ただし、長さ 1 の carray 型と char 型は、以下の点で異なります。

  - char 型には、関連する fieldid のオーバーヘッド分が含まれますが、carray 型には、関連する fieldid のほか、長さコードが含まれます。

- carray 型を数値型に変換する場合は、string 型にしてから atoi を呼び出します。char を型変換すると、数値になります。たとえば、ASCII 値「1」(10 進数 では 49) の char 型は、値 49 の short 型に変換されます。長さ 1 の carray 型 (単一バイトの ASCII 値「1」を持つ) は、値 1 の short 型に変換されます。同様に、char 型の「a」(10 進数では 97) は、値 97 の short 型に変換されます。carray 型の「a」は、atoi (「a」) が 0 を生成するので、値 0 の short 型に変換されます。

■ dec_t 型を別の型に変換したり、またはその逆処理を行う場合、decimal(3c) で説明されている変換関数 (_gp_deccvasc、_gp_deccvdbl、_gp_deccvflt、_gp_deccvint、_gp_deccvlong、_gp_dectoasc、_gp_dectodbl、_gp_dectoflt、_gp_dectoint、および _gp_dectolong) を使用します。

次の表は、この節で示した変換規則をまとめたものです。

表 5-2 変換規則のまとめ

変換前の型	変換後の型							
-	char	short	long	float	double	string	carray	dec_t
char	-	cast	cast	cast	cast	st[0]=c	array[0]=c	d
short	cast	-	cast	cast	cast	sprintf	sprintf	d
long	cast	cast	-	cast	cast	sprintf	sprintf	d
float	cast	cast	cast	-	cast	sprintf	sprintf	d
double	cast	cast	cast	cast	-	sprintf	sprintf	d
string	c=st[0]	atoi	atol	atof	atof	-	drop 0	d
carray	c=array[0]	atoi	atol	atof	atof	add 0	-	d
dec_t	d	d	d	d	d	d	d	-

次の表は、前の表で使用されているエントリの説明です。

表 5-3 エントリの説明

エントリ	説明
-	src と dest が同じ型です。変換の必要はありません。
cast	C の代入演算子を使用した型変換によって、変換が行われます。
sprintf	sprintf 関数によって変換が行われます。
atoi	atoi 関数によって変換が行われます。
atof	atof 関数によって変換が行われます。
atol	atol 関数によって変換が行われます。
add 0	NULL バイトの連結によって変換が行われます。
drop 0	NULL 終了バイトを削除することによって変換が行われます。
c=array[0]	文字が配列の最初のバイトに設定されます。
array[0]=c	配列の最初のバイトが文字に設定されます。
c=st[0]	文字が文字列の最初のバイトに設定されます。
st[0]=c	文字列の最初のバイトが文字に設定されます。
d	decimal(3c) 変換関数

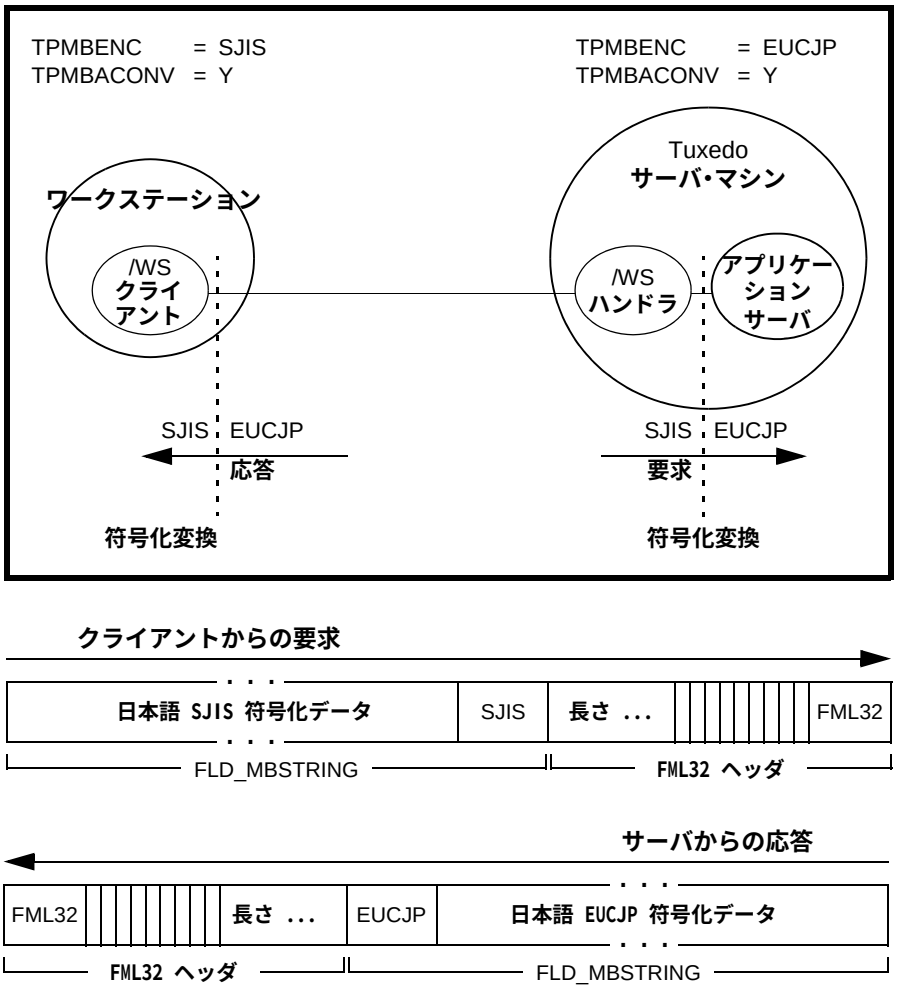
FLD_MBSTRING フィールドの変換

ユーザ指定の型 FLD_MBSTRING のデータのコード・セットの符号化変換を処理するために、以下の関数が提供されています。

- Fmbpack32(3fml)
- Fmbunpack32(3fml)
- tpconvfmb32(3fml)

これらの関数では、FLD_MBSTRING フィールドの符号化名とマルチバイト・データ情報の準備、FLD_MBSTRING フィールドからの符号化名とマルチバイト・データ情報の抽出、および FLD_MBSTRING フィールドにあるマルチバイト文字の名前付きターゲット符号化への変換を行います。次の図は、符号化変換がどのように行われるかを例示しています。

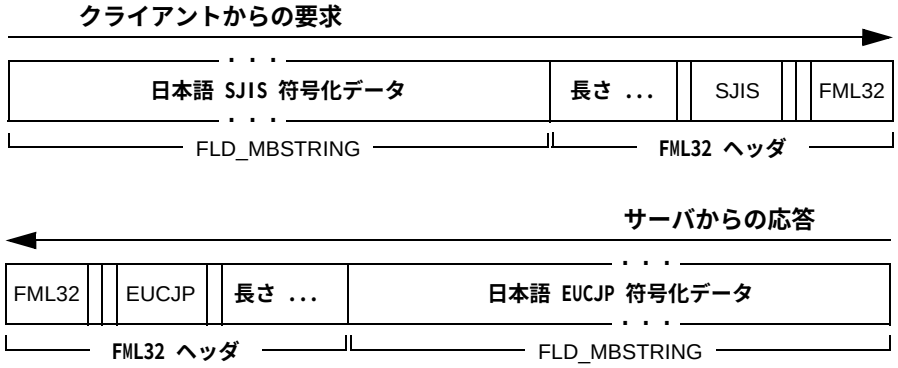
図 5-1FML32 バッファによる符号化変換の例



上図の例に示すように、FLD_MBSTRING フィールドは、ユーザ・データのコード・セット文字符号化 (または単に符号化) を識別する情報を保持できます。この例では、クライアントの要求の FLD_MBSTRING フィールドは、Shift-JIS (SJIS) 符号化で表される日本語ユーザ・データを保持し、サーバの応答の FLD_MBSTRING フィールドは、Extended UNIX Code (EUC) 符号化で表される日本語ユーザ・データを保持しています。マルチバイト文字符号化機能では、環境変数 TPMBENC と TPMBACONV を読み込んで、ソースの符号化、ターゲットの符号化、および自動符号化変換の状態 (有効または無効) を判別します。

次の図に示すように、FML32 型付きバッファは、それ自体でユーザ・データの文字符号化を識別する情報を保持できます。

図 5-2 グローバル符号化の使用



FML32 型付きバッファで保持する FLD_MBSTRING フィールド数が多い場合、グローバル符号化を使用すると、FLD_MBSTRING フィールドごとに文字符号化名を追加するよりも、FML32 バッファによるマルチバイト・ユーザ・データの転送効率が向上します。Fmbpack32() 関数を使用すると、アプリケーション開発者は、Fmbpack32() で作成された FLD_MBSTRING フィールドごとに、グローバル符号化を行うか個別の符号化を行うかを選択できます。グローバル符号化で使用できる名前は、各 FML32 バッファにつき 1 つだけです。

符号化の変換機能により、基盤となる Tuxedo システム・ソフトウェアでは、着信 FLD_MBSTRING フィールドの符号化表現を、受信プロセスが実行されているマシンでサポートされている符号化表現に変換できます。この変換は文字コード・セット間の変換でも言語の翻訳でもなく、同じ言語の異なる文字符号化間の変換です。

Fmbpack32

この関数は、FML32 型付きバッファに入力された FLD_MBSTRING フィールドの符号化名およびマルチバイト・データ情報を準備します。Fmbpack32() は、FLD_MBSTRING フィールドが FML32 API で FML32 バッファに追加される前に使用します。

この関数の詳細については、『BEA Tuxedo FML リファレンス』の Fmbpack32(3fml) 関数を参照してください。

Fmbunpack32

この関数は、FML32 型付きバッファ内の FLD_MBSTRING フィールドから符号化名およびマルチバイト・データ情報を抽出します。Fmbunpack32() は、FLD_MBSTRING フィールドが FML32 API (Ffind32()、Fget32()、など) で FML32 バッファから抽出された後に使用します。

この関数の詳細については、『BEA Tuxedo FML リファレンス』の Fmbunpack32(3fml) 関数を参照してください。

tpconvfmb32

この関数は、FML32 型付きバッファにある FLD_MBSTRING フィールドのマルチバイト文字を、ターゲットの名前付きの符号化に変換します。具体的には、tpconvfmb32() は、FLD_MBSTRING フィールドで指定されたソースの符

号化名と *target_encoding* で定義されたターゲットの符号化名を比較し、符号化名が異なる場合に tpconvfmb32() は、FLD_MBSTRING フィールドのデータをターゲットの符号化に変換します。

この関数の詳細については、『BEA Tuxedo FML リファレンス』の tpconvfmb32(3fml) 関数を参照してください。

インデックスを操作する関数

フィールド化バッファを Finit または Falloc で初期化すると、自動的にインデックスが設定されます。このインデックスにより、フィールド化バッファへのアクセスが促進されますが、プログラマ側からはインデックス処理が見えません。フィールド化バッファにフィールドを追加したり、削除すると、インデックスが自動的に更新されます。

ただし、記憶装置に長期に渡ってフィールド化バッファを格納したり、協調動作するプロセス間でフィールド化バッファを転送する場合は、フィールド化バッファの受信時に、インデックスを削除したり、再生成して領域を節約できます。ここで説明する関数は、このようなインデックス操作を実行します。

Fidxused

Fidxused は、バッファのインデックスが占有する領域を返します。

```
long
Fidxused(FBFR *fbfr)
```

fbfr は、フィールド化バッファに対するポインタです。

この関数を使用すると、バッファのインデックスのサイズを判別し、インデックスを削除した方が時間や領域を節約できるかどうかを判別できます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fidxused、Fidxused32(3fml)」を参照してください。

Findex

Findex を使用すると、インデックスが設定されていないフィールド化バッファにインデックスを設定できます。

```
int
Findex(FBFR *fbfr.FLDOCC intvl)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *intvl* は、インデックス付けの間隔です。

Findex の 2 つ目の引数は、バッファのインデックス付けの間隔を指定します。0 を指定すると、FSTDXINT 値 (fml.h で定義) が使用されます。間隔 1 を指定すると、すべてのフィールドにインデックスが設定されます。

インデックス付けの間隔を増やしてバッファのインデックスを再設定すると、ユーザ・データ用のバッファの空き領域を増やすことができます。ただし、処理時間の高速化と空き領域の確保を一度に両方実現することはできません。一般的に、インデックスの数を減らす、つまりインデックスの間隔を大きくすると、フィールドの検索に時間がかかります。ほとんどの操作では、空き領域が少なくなると、まず、すべてのインデックスが削除されます。それができない場合は、エラー・メッセージが返されます。

詳細については、『BEA Tuxedo FML リファレンス』の「Findex、Findex32(3fml)」を参照してください。

Frstrindex

インデックスの削除後に、フィールド・バッファを変更していない場合は、この関数を Findex の代わりに使用できます。

```
int
Frstrindex(FBFR *fbfr, FLDOCC numidx)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。

- *numidx* は、Funindex 関数の戻り値です。

詳細については、『BEA Tuxedo FML リファレンス』の「Frstrindex、Frstrindex32(3fml)」を参照してください。

Funindex

Funindex は、フィールド化バッファのインデックスを削除し、削除前のバッファ内のインデックス数を返します。

```
FLDOCC
Funindex(FBFR *fbfr)
```

fbfr は、フィールド化バッファに対するポインタです。

詳細については、『BEA Tuxedo FML リファレンス』の「Funindex、Funindex32(3fml)」を参照してください。

インデックスを設定しないでフィールド化バッファを送信する例

インデックスを設定しないでフィールド化バッファを送信するには、次の手順に従います。

1. インデックスを削除します。

   ```
   save = Funindex(fbfr);
   ```

2. 送信するバイト数 (バッファの先頭の最上位バイトの数) を取得します。

   ```
   num_to_send = Fused(fbfr);
   ```

3. インデックスが設定されていないバッファを送信します。

   ```
   transmit(fbfr,num_to_send);
   ```

4. バッファのインデックスを復元します。

   ```
   Frstrindex(fbfr,save);
   ```

インデックスは、次の文を使用して受信側で再生成できます。

```
Findex(fbfr);
```

インデックスの削除は受信側のプロセス以外で行われ、さらにファイルで送信されなかったため、受信側のプロセスは Frstrindex を呼び出せません。

注記　Funindex を呼び出しても、インデックスが占有していたメモリ領域は解放されません。Funindex は、ディスク上の領域を節約したり、バッファを別プロセスに送信するときに領域を節約するだけです。フィールド化バッファとインデックスを別プロセスに送信し、これらの関数を使用しないこともできます。

入出力を操作する関数

この節では、標準入出力またはファイルのストリームに対して、フィールド化バッファの入力や出力を行うための関数を説明します。

Fread および Fwrite

I/O 関数の Fread および Fwrite は、標準 I/O ライブラリを操作します。

```
int Fread(FBFR *fbfr, FILE *iop)
int Fwrite(FBFR *fbfr, FILE *iop)
```

入出力先のストリームは、FILE 型のポインタ引数によって決定されます。この引数は、通常の標準 I/O ライブラリ関数を使用して設定しなければなりません。

以下のように Fwrite を使用すると、フィールド化バッファを標準 I/O ストリームに書き込むことができます。

```
if (Fwrite(fbfr, iop) < 0)
  F_error("pgm_name");
```

以下のように Fread を使用すると、Fwrite を書き込んだバッファを読み取ることができます。

```
if(Fread(fbfr, iop) < 0)
 F_error("pgm_name");
```

fbfr が指すフィールド化バッファの内容は、読み込まれたフィールド化バッファの内容で置き換えられますが、フィールド化バッファの容量 (バッファ・サイズ) は変更されません。

Fwrite は、バッファのインデックスを削除し、Fused から返されたフィールド化バッファの使用済みの部分のみを書き込みます。

Fread は、Findex を呼び出すことによってバッファのインデックスを復元します。バッファのインデックス付けには、Fwrite で書き込まれた時と同じインデックス付けの間隔が使用されます。Fread32 は、FLD_PTR 型のフィールドを無視します。

詳細については、『BEA Tuxedo FML リファレンス』の「Fread、Fread32(3fml)」および「Fwrite、Fwrite32(3fml)」を参照してください。

Fchksum

以下のようにチェックサムを算出して、I/O の妥当性をチェックできます。

```
long chk;
. . .
chk = Fchksum(fbfr);
```

Fchksum を呼び出し、フィールド化バッファと共にチェックサムの値を書き出し、入力時にその値をチェックする処置は、ユーザの責任です。Fwrite は、チェックサムの自動的な書き込みは行いません。ポインタ・フィールド (FLD_PTR) の場合は、ポインタまたはポインタが参照するポインタではなく、チェックサムの計算に使用するポインタ・フィールド名が含まれます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fchksum、Fchksum32(3fml)」を参照してください。

Fprint および Ffprint

Fprint は、フィールド化バッファをテキスト形式で標準出力に出力します。

```
Fprint(FBFR *fbfr)
```

fbfr は、フィールド化バッファに対するポインタです。

Ffprint は、Fprint とよく似ていますが、次の例のように、テキストを指定された出力ストリームに出力します。

```
Ffprint(FBFR *fbfr, FILE *iop)
```

以下はパラメータの説明です。

- *fbfr* は、フィールド化バッファに対するポインタです。
- *iop* は、出力ストリームへの FILE 型ポインタです。

これらの各出力関数は、フィールド・オカレンスごとに、フィールド名とフィールド値をタブで区切って出力し、復帰改行文字を追加します。Fname は、フィールド名の判別に使用しますが、フィールド名を判別できないと、フィールド識別子を出力します。文字列または文字配列のフィールド値内の表示不能な文字は、バックスラッシュとその後に続く 2 文字からなる 16 進値で表現されます。テキスト内に現れるバックスラッシュは、もう 1 つバックスラッシュを使用するとエスケープされます。バッファの出力が完了すると空白行が出力されます。

値が FLD_PTR 型の場合、Fprint32 はフィールド名またはフィールド識別子と 16 進法のポインタ値を出力します。この関数はポインタ情報を出力しますが、Fextread32 関数はフィールド型 FLD_PTR を無視します。値が FLD_FML32 型の場合、Fprint32 は、入れ子の各レベルの先頭にタブを付けて FML32 バッファを再帰的に出力します。値が FLD_VIEW32 型の場合、この関数は、VIEW32 フィールド名と構造体メンバ名 / 値の対を出力します。

詳細については、『BEA Tuxedo FML リファレンス』の「Fprint、Fprint32(3fml)」を参照してください。

Fextread

Fextread を使用すると、フィールド化バッファを、その出力形式、つまり Fprint の出力から作成できます (Fprint で出力された 16 進値が正しく解釈されます)。

```
int
Fextread(FBFR *fbfr, FILE *iop)
```

Fextread は、Fprint の出力形式で、フィールド名とフィールド識別子の組み合わせの前指定する、次のオプション・フラグを受け付けます。

表 5-4Fextread のフラグ

フラグ	指定内容
+	バッファ内でフィールドを変更します。
-	バッファからフィールドを削除します。
=	フィールドを別のフィールドに代入します。
#	コメント行です。これは無視されます。

フラグが指定されていない場合、デフォルトでは、Fadd によりフィールドが追加されます。

複数行にわたってフィールド値を指定するには、2 行目以降の行頭にタブを入力します。このタブは無視されます。単一の空白行は、バッファの終了を示します。連続した複数の空白行を指定すると、NULL バッファが生成されます。埋め込み型のバッファに対して Fextread を実行すると、入れ子状の FML32 バッファ (FLD_FML32) と VIEW32 フィールド (FLD_VIEW32) が生成されます。Fextread32 は、FLD_PTR 型のフィールドを無視します。

エラーが発生すると、-1 が返され、Ferror が適宜設定されます。ファイルの終わりに達しても空白行が現れないと、Ferror に FSYNTAX が設定されます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fextread、Fextread32(3fml)」を参照してください。

フィールド化バッファの論理式

ここでは、次の内容について説明します。

- 論理式の定義
- フィールド名とフィールド型
- 論理式を評価に適した形式に変換する方法
- 論理式の基本式

この節では、式の「変数」にフィールド化バッファまたは VIEW のフィールドの値であるような論理式を評価する関数について説明します。これらの関数を使用すると、以下の処理を行うことができます。

- 論理式を、評価に適したコンパクトな形式にコンパイルします。
- フィールド化バッファまたは VIEW と照合して論理式を評価し、true または false を返します。
- コンパイル済みの論理式を出力します。

ここでは、論理式をコンパクトな形式にコンパイルして効率的に評価するための関数と、コンパイル済みの論理式をフィールド化バッファと照合させ、true または false を返す関数を説明します。

論理式の定義

ここでは、論理式のコンパイル用関数で受け付ける式と、式の評価方法を詳しく説明します。

以下の C 言語の標準演算子はサポートされていません。

- シフト演算子 (<<、>>)
- ビット単位の論理和演算子 (||) および 論理積演算子 (&&)

- 条件演算子 (?)
- 前置または後置のインクリメント演算子 (++) およびデクリメント演算子 (--)
- アドレス演算子 (&) および間接演算子 (*)
- 代入演算子 (=)
- カンマ演算子 (,)

次の表は、論理式で受け付けられる BNF 記法の一覧です。

表 5-5BNF 記法による論理式の定義（シート 1 / 2）

式	定義
<boolean>	<boolean> || <logical and> | <logical and>
<logical and>	<logical and> && <xor expr> | <xor expr>
<xor expr>	<xor expr> ^ <equality expr> | <equality expr>
<equality expr>	<equality expr> <eq op> <relational expr> | <relational expr>
<eq op>	== | != | %% | !%
<relational expr>	<relational expr> <rel op> <additive expr> | <additive expr>
<rel op>	< | <= | >= | > |
<additive expr>	<additive expr> <add op> <multiplicative expr> | <multiplicative expr>
<add op>	+ | -
<multiplicative expr>	<multiplicative expr> <mult op> <unary expr> | <unary expr>
<mult op>	* | / | %
<unary expr>	<unary op> <primary expr> | <primary expr>
<unary op>	+ | - | ~ | !
<primary expr>	(<boolean>) | <unsigned constant> | <field ref>
<unsigned constant>	<unsigned number> | <string>

表 5-5BNF 記法による論理式の定義（シート 2 / 2）

式	定義
<unsigned number>	<unsigned float> | <unsigned int>
<string>	' <character> {<character>. . .} '
<field ref>	<field name> | <field name>[<field occurrence>]
<field occurrence>	<unsigned int> | <meta>
<meta>	?

以降の節では、論理式についてさらに詳しく説明します。

フィールド名とフィールド型

論理式で許可される変数は、フィールド参照のみです。フィールド名の指定方法には、いくつかの規則があります。たとえば、フィールド名は、英字と数字で構成し、先頭には英字を指定しなければなりません。下線 (_) は、英字と見なされます。長い変数名を下線で区切り、読みやすい名前に変えることができます。フィールド名には、最大 30 文字を指定できます。予約語はありません。

フィールド化バッファを評価するため、論理式で参照されるフィールドは、フィールド・テーブルに存在していなければなりません。したがって、3-1 ページの「FML および VIEWS の環境設定」で説明したように、環境変数の `FLDTBLDIR` と `FIELDTBLS` を設定してから、論理式をコンパイルする関数を使用してください。論理式で使用できるフィールド型は、FML フィールドで使用できる型と同じです。つまり、`short` 型、`long` 型、`float` 型、`double` 型、`char` 型、`string` 型、および `carray` 型を使用できます。フィールド型は、フィールド名と共にフィールド・テーブルに格納されています。したがって、フィールド型は常に判別できます。

VIEW を評価するため、論理式で参照されるフィールドは、対応するフィールド化バッファ名ではなく、C 構造体の要素名として VIEW に存在していなければなりません。したがって、3-1 ページの「FML および VIEWS の環境設定」で説明したように、環境変数の VIEWDIR と VIEWFILES を設定してから、論理式をコンパイルする関数を使用してください。論理式で使用できるフィールド型は、FML VIEWS で使用できる型と同じです。つまり、short 型、long 型、float 型、double 型、char 型、string 型、carray 型のほか、int 型および dec_t 型を使用できます。フィールド型は、フィールド名と共に VIEW 定義に格納されています。したがって、フィールド型は常に判別できます。

文字列

文字列は、一重引用符で囲まれた文字の集まりです。エスケープ・シーケンスでエスケープした文字は、その文字の ASCII コードで置き換えることができます。エスケープ・シーケンスは、バックスラッシュと 2 桁の 16 進数で構成されます。この規則は、\x で始める 16 進のエスケープ・シーケンスを使用する C 言語規則とは異なります。

たとえば、'hello' と 'hell\\6f' の場合、'o' の 16 進数は 6f なので、これらの文字列は同一と見なされます。

8 進のエスケープ・シーケンスや \n などのエスケープ・シーケンスはサポートされていません。

定数

定数として、C 言語の場合と同じく、整数と浮動小数点値が受け付けられます。8 進および 16 進の定数は認識されません。整数は、long 型として処理され、浮動小数点値は、double 型として処理されます。(dec_t 型の 10 進定数はサポートされていません。)

論理式を評価に適した形式に変換する方法

論理式を評価するため、論理式のコンパイラによって以下の変換が行われます。

- short 型および int 型の値を long 型に変換します。
- float 型および decimal 型の値を double 型に変換します。
- 文字を string 型に変換します。
- フィールド内の引用符で囲まれていない文字列を数値と比較するため、文字列を数値に変換します。
- 引用符で囲まれた constant の文字列を数値と比較するため、数値を文字列に変換してから、字句単位で比較します。
- long 型と double 型を比較するため、long 型を double 型に変換します。

論理式の基本式

論理式は、以下の基本式で構成されています。

- field name －フィールド名
- field name[constant] －フィールド名と定数添字
- field name[?] －フィールド名と ‘?‘ 添字
- constant －定数
- (expression) －かっこで囲まれた式

フィールド名またはフィールド名と後続の添字は、基本式です。添字は、参照対象のフィールド・オカレンスを指定します。添字としては、整定数または ? (任意のオカレンスを示す) のいずれかを使用します。添字は、式と見なされません。フィールド名に添字が付いていないと、フィールド・オカレンスは 0 と見なされます。

フィールド名参照が、算術演算子、単項演算子、代入演算子、または関係演算子なしで示されると、フィールドが存在する場合は long 型整数の 1、フィールドが存在しない場合は 0 となります。この方法を使用して、フィールド型に関係なく、フィールド化バッファ内にフィールドが存在するかどうかをテストできます。間接演算子 (*) は存在しません。

定数は基本式です。定数の型は、`long` 型、`double` 型、`carray` 型のいずれかです。型変換の説明を参照してください。

かっこで囲んだ式は、型と値が、かっこで囲まない場合の式の型と値に等しい基本式です。かっこを使用すると、演算子の優先順位を変更できます。次の節を参照してください。

論理式の演算子

次の表は、論理式の演算子を優先順位の高いものから順に示しています。

表 5-6 論理式の演算子

タイプ	演算子
単項	+, -, !, ~
倍数	*, /, %
加法	+, -
比較	< , >, <=, >=, ==, !=
等価、一致	==, !=, %%, !%
排他論理和	^
論理積	&&

表 5-6 論理式の演算子

タイプ	演算子
論理和	||

同じ演算子型に分類される演算子は、優先順位が同じです。次の節では、各演算子を詳しく説明します。C 言語では、かっこを使用して、演算子の優先順位を上書きできます。

論理式で使用される単項演算子

以下の単項演算子が認識されます。

- 単項プラス演算子 (+)
- 単項マイナス演算子 (-)
- 補数演算子 (~)
- 論理否定演算子 (!)

単項演算子を含む式は、右から左にグループ化されます。

```
+ expression
- expression
~ expression
! expression
```

単項プラス演算子は、オペランドに有効ではありません。認識はされても、無視されます。単項マイナス演算子の結果は、そのオペランドの否定です。通常の算術変換が実行されます。符号なしのものは、FML には存在しないので、この演算子には問題がありません。

論理否定演算子の結果は、オペランドの値が 0 の場合は 1 であり、オペランドの値が 0 以外の場合は 0 となります。結果は `long` 型になります。

補数演算子の結果は、オペランドの補数です。結果は `long` 型になります。

論理式で使用される倍数演算子

倍数に関する演算子の *、/、% は、左から右にグループ化されます。通常の算術変換が実行されます。

```
expression * expression
expression / expression
expression % expression
```

2 項演算子 * は、乗算を示します。* 演算子は、連想型であり、同一レベルで数回の乗算を行う式は、コンパイラで再配置できます。

2 項演算子 / は、除算を示します。正の整数が除算されると、切り捨ては 0 に向かって行われますが、オペランドのいずれかが負の場合は、切り捨ての形式は、マシンによって異なります。

2 項演算子 % は、最初の式を次の式で除算した結果の剰余を返します。通常の算術変換が実行されます。オペランドには、float 型や double 型を使用してはなりません。

論理式で使用される加法演算子

加法に関する演算子の + と - は、左から右にグループ化されます。通常の算術変換が実行されます。

```
expression + expression
expression - expression
```

+ 演算子は、オペランドの和を返します。+ 演算子は連想型であり、同一レベルで数回の加算を行う式は、コンパイラで再配置できます。オペランドが両方とも string である必要はありません。一方のオペランドが string 文字列の場合、そのオペランドはもう一方のオペランドの算術型に変換されます。

- 演算子は、オペランドの差を返します。通常の算術変換が実行されます。オペランドが両方とも string である必要はありません。一方のオペランドが string 文字列の場合、そのオペランドはもう一方のオペランドの算術型に変換されます。

論理式で使用される等価、一致に関する演算子

この型の演算子は、左から右にグループ化されます。

```
expression == expression
expression != expression
expression %% expression
expression !% expression
```

== (等価)、!= (非等価) の各演算子は、指定された関係が false の場合は 0 を返し、true の場合は 1 を返します。結果は long 型になります。通常の算術変換が実行されます。

%% 演算子の場合は、2 つ目の式は、最初の式と照合するために使用する正規表現です。2 つ目の式 (正規表現) は、引用符で囲まれた文字列でなければなりません。最初の式は、FML フィールド名であっても、引用符で囲まれた文字列であってもかまいません。この演算子は、最初の式が 2 つ目の式 (正規表現) と完全に一致する場合は、1 を返し、それ以外は 0 を返します。

!% 演算子は、正規表現との不一致を検出する演算子です。この演算子は、%% 演算子と同じオペランドをとりますが、まったく反対の結果を生じます。%% と !% の関係は、== と != の関係と同じです。

使用可能な正規表現については、『BEA Tuxedo C リファレンス』の「tpsubscribe(3c)」リファレンス・ページを参照してください。

論理式で使用される関係演算子

この型の演算子は、左から右にグループ化されます。

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

< (より小さい)、> (より大きい)、<= (以下)、>= (以上) の各演算子は、指定された関係が false であれば 0 を返し、true であれば 1 を返します。結果は long 型になります。通常の算術変換が実行されます。

論理式で使用される排他論理和演算子

演算子 ^ は、左から右にグループ化されます。

expression ^ *expression*

ビット単位の排他論理和が返されます。結果は、必ず long 型です。

論理式で使用される論理積演算子

expression && *expression*

&& 演算子は、左から右にグループ化されます。この演算子は、両方のオペランドが 0 でない場合は 1 を返し、どちらかが 0 の場合は 0 を返します。&& 演算子は、必ず左から右へ評価を行います。ただし、最初のオペランドが 0 の場合、2 つ目のオペランドを評価しないということはなく、この点で C 言語とは異なります。オペランドの型は同一でなくてもかまいません。結果は、必ず long 型です。

論理式で使用される論理和演算子

|| 演算子は、左から右にグループ化されます。

expression || *expression*

どちらかのオペランドが 0 でない場合は 1 を返し、両方とも 0 の場合は 0 を返します。|| 演算子は、必ず左から右へ評価を行います。ただし、最初のオペランドが 0 でない場合、2 つ目のオペランドを評価しないということはなく、この点で C 言語と異なります。オペランドの型は同一でなくてもかまいません。結果は、必ず long 型です。

論理式のサンプル

以下のフィールド・テーブルは、論理式のサンプルで使用するフィールドを定義しています。

```
EMPID    200    carray
SEX      201    char
AGE      202    short
DEPT     203    long
SALARY   204    float
NAME     205    string
```

論理式は、必ず true か false に評価されることに注意してください。次の例では、条件が両方とも true の場合、true になります。

- EMPID のフィールド・オカレンス 2 が存在し、その値が文字「123」で始まる。
- AGE フィールド (オカレンス 0) が存在し、その値が 32 未満である。

```
"EMPID[2] %% '123.*' && AGE < 32"
```

この例では、EMPID の添字として整定数を使用しています。以下の例では、? 添字を使用しています。

```
"PETS[?] == 'dog'"
```

この式は、PETS が存在し、その任意のオカレンスが文字「dog」を含む場合は、true となります。

論理式を処理する関数

ここでは、引数として論理式をとる各種の関数について説明します。

Fboolco および Fvboolco

Fboolco は、論理式をコンパイルし、評価ツリーに対するポインタを返します。

```
char *
Fboolco(char *expression)
```

*expression はコンパイル対象の式に対するポインタです。次のいずれかのフィールド・タイプが使用されると、この関数は異常終了します。FLD_PTR、FLD_FML32、または FLD_VIEW32。これらのフィールド・タイプの 1 つが指定されると、Ferror に FEBADOP が設定されます。

Fvboolco は、VIEW の論理式をコンパイルし、評価ツリーに対するポインタを返します。

```
char *
Fvboolco(char *expression, char *viewname)
```

*expression はコンパイル対象の式に対するポインタで、*viewname はフィールドを評価する VIEW 名に対するポインタです。

評価ツリーを保持するには、malloc(3) を使用して領域を割り当てます。たとえば、次の例は、「J」で始まり「n」で終了する FIRSTNAME フィールド (たとえば、「John」、「Joan」など) がバッファ内に存在し、かつ、SEX (性別) フィールドが「M」に設定されているかどうかをチェックする論理式をコンパイルします。

```
#include "<stdio.h>"
#include "fml.h"
extern char *Fboolco;
char *tree;
. . .
if((tree=Fboolco("FIRSTNAME %% 'J.*n' && SEX == 'M'")) == NULL)
  F_error("pgm_name");
```

ツリー配列の最初と 2 つ目の文字は、それぞれ、配列全体の長さをビット単位で提供する unsigned 型の 16 ビット量の最下位バイトと最上位バイトを形成します。この値は、配列のコピーなどの操作を行う場合に有用です。

Fboolco が生成する評価ツリーは、次の節で説明する論理式関数で使用されます。したがって、絶えず式の再コンパイルをする必要はありません。

論理式を使用する必要がなくなった時に評価ツリーに割り当てられた領域を解放するには、free(3) を使用してください。必要がなくなった評価ツリーを解放せずに多くの論理式をコンパイルすると、プログラムのデータ領域がなくなってしまう恐れがあります。

詳細については、『BEA Tuxedo FML リファレンス』の「Fboolco、Fboolco32、Fvboolco、Fvboolco32(3fml)」を参照してください。

Fboolpr および Fvboolpr

Fboolpr は、指定されたファイルにコンパイルされた式を出力します。コンパイルされた式は、構文解析された (評価ツリーで示された) ようにコンパイルされ、全体にかっこが付きます。

```
void
Fboolpr(char *tree, FILE *iop)
```

以下はパラメータの説明です。

- *tree は、Fboolco によって以前にコンパイルされた論理式ツリーに対するポインタです。
- *iop は、出力ファイルへの FILE 型ポインタです。

Fvboolpr は、指定されたファイルにコンパイルされた式を出力します。

```
void
Fvboolpr(char *tree, FILE *iop, char *viewname)
```

以下はパラメータの説明です。

- *tree は、Fvboolco を使用して以前にコンパイルした論理式ツリーに対するポインタです。
- *iop は、出力ファイルへの FILE 型ポインタです。
- *viewname は、フィールドが使用されている VIEW の名前です。

この関数は、デバッグを行う場合に有用です。

Fboolco の例でコンパイルされた式を Fboolpr にかけると、以下の結果が出力されます。

```
(((FIRSTNAME[0]) %% ('J.*n')) && ((SEX[0]) == ('M')))
```

詳細については、『BEA Tuxedo FML リファレンス』の「Fboolpr、Fboolpr32、Fvboolpr、Fvboolpr32(3fml)」を参照してください。

Fboolev と Ffloatev、および Fvboolev と Fvfloatev

これらの関数は、両方とも、フィールド化バッファと照合して論理式を評価します。

```
int Fboolev(FBFR *fbfr,char *tree)
double Ffloatev(FBFR *fbfr,char *tree)
```

以下はパラメータの説明です。

- *fbfr* は、Fboolco によって生成された評価ツリーが参照するフィールド化バッファです。
- *tree* は、fbfr が指すフィールド化バッファを参照する評価ツリーに対するポインタです。

同等の VIEW の関数を以下に示します。

```
int
Fvboolev(FBFR *fbfr,char *tree,char *viewname)

double
Fvfloatev(FBFR *fbfr,char *tree,char *viewname)
```

Fboolev は、フィールド化バッファが評価ツリーで指定された論理式の条件と一致する場合は true (1) を返します。この関数は、フィールド化バッファあるいは評価ツリーのいずれも変更しません。上記の例のコンパイルされた評価ツリーを使用すると、「Buffer selected」と出力されます。

```
#include <stdio.h>
#include "fml.h"
#include "fldtbl.h"
FBFR *fbfr;
. . .
Fchg(fbfr,FIRSTNAME,0,"John",0);
Fchg(fbfr,SEX,0,"M",0);
```

```
if(Fboolev(fbfr,tree) > 0)
  fprintf(stderr,"Buffer selected\n");
else
  fprintf(stderr,"Buffer not selected\n");
```

Ffloatev および Ffloatev32 は、Fboolev と同様に動作しますが、式の値を double 型として返します。たとえば、次のコードでは、「6.6」と出力されます。

```
#include <stdio.h>
#include "fml.h"
FBFR *fbfr;
. . .
main() {
  char *Fboolco;
  char *tree;
  double Ffloatev;
  if (tree=Fboolco("3.3+3.3")) {
      printf("%lf",Ffloatev(fbfr,tree));
  }
}
```

Fboolev を上記の例の Floatev の位置で使用した場合、1 が出力されます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fboolev、Fboolev32、Fvboolev、Fvboolev32(3fml)」および「Ffloatev、Ffloatev32、Fvfloatev、Fvfloatev32(3fml)」を参照してください。

VIEW の変換

VIEW を、指定したレコード形式に変換したり、その逆の処理を行うことができます。デフォルトでは、IBM System/370 COBOL レコード形式に変換されます。

Fvstot、Fvftos、および Fcodeset

以下の関数は、指定した形式に VIEW を変換します。

```
long
Fvstot(char *cstruct, char *trecord, long treclen, char *viewname)

long
Fvttos(char *cstruct, char *trecord, char *viewname)

int
Fcodeset(char *translation_table)
```

Fvstot 関数は、C 構造体のデータを、指定したレコードの形式に転送します。Fvttos 関数は、指定レコード形式のデータを C 構造体に転送します。trecord は指定したレコードに対するポインタです。*cstruct* は C 構造体に対するポインタです。viewname は、コンパイルされた VIEW 記述名に対するポインタです。VIEWDIR 環境変数および VIEWFILES 環境変数を使用してコンパイルされた VIEW 記述があるディレクトリとファイルが検索されます。

FML バッファを、指定したレコードに変換するには、次の手順に従います。

1. Fvftos を呼び出して FML バッファを C 構造体に変換します。
2. Fvstot を呼び出して指定レコードに変換します。

指定レコードを、FML バッファに変換するには、次の手順に従います。

1. Fvttos を呼び出して C 構造体に変換します。
2. Fvstof を呼び出してその構造体を FML バッファに変換します。

デフォルトは、IBM/370 COBOL レコードの形式です。デフォルトのデータ変換は、以下の表に基づいて行われます。

表 5-7 構造体からレコードへの変換

データ構造	レコード
float	COMP-1
double	COMP-2
long	S9(9) COMP
short	S9(4) COMP
int	S9(9) COMP または S9(4) COMP
dec_t(*m, n*)	S9(*2*m*-(*n*+*1*))V9(*n*)COMP-3
ASCII char	EBCDIC char
ASCII string	EBCDIC string
carray	文字配列

IBM/370 のレコードでは、フィールド間のフィルタ・バイトはありません。ビューに対応するデータ構造の一部をなすデータ・アイテムに対しては、COBOL SYNC 節は指定することはできません。整数フィールドは、変換を実行するマシン上の整数のサイズに応じて 4 バイトまたは 2 バイトの整数に変換されます。IBM/370 フォーマットへの変換、または IBM/370 フォーマットからの変換を行う場合、ビューの文字列フィールドは NULL で終了している必要があります。carray フィールドのデータは変更されずに渡されます。データの変換は行われません。

パック 10 進数は、IBM/370 環境では 1 バイトにパッキングされた 2 桁の 10 進数として存在し、下位の 1/2 バイトは符号を格納するために使用されます。パッキングされた 10 進数の長さは 1 ～ 16 バイトで、1 ～ 31 桁の数字と 1 つの符号の格納領域があります。パッキングされた 10 進数は、C の構造体では dec_t というフィールド・タイプを利用することによってサポートされます。dec_t フィールドは、カンマで区切られた 2 つの数字で構成されたサイズに定義されています。カンマの左側の数字は、10 進数が占有する総バイト数です。右側の数値は、小数点以下の桁数です。変換には、次の公式が使用されます。

```
dec_t(m, n) <=> S9(2*m-(n+1))V9(n)COMP-3
```

10 進数とほかのデータ型 (int 型、long 型、string 型、double 型、float 型) 間の変換には、decimal(3c) で説明されている関数を使用できます。

ASCII から EBCDIC へ、また EBCDIC から ASCII へのデフォルトの文字変換の詳細については、「Fvstof、Fvstof32(3fml)」の項を参照してください。

Fcodeset を呼び出すと、実行時に代替文字変換テーブルを使用することができます。translation_table は、512 バイトのバイナリ・データを指している必要があります。最初の 256 バイトのデータは、ASCII から EBCDIC への変換テーブルとして解釈されます。次の 256 バイトのデータは、EBCDIC から ASCII へのテーブルとして解釈されます。512 バイトより後ろのデータは無視されます。ポインタが NULL のときは、デフォルトの変換テーブルが使用されます。

詳細については、『BEA Tuxedo FML リファレンス』の「Fvstot、Fvttos(3fml)」を参照してください。

6 FML および VIEWS の例

ここでは、次の内容について説明します。

- VIEWS の使用例
- bankapp での FML の使用例

VIEWS の使用例

この節で示す VIEWS の使用例は、この章の後半にある FML プログラムの使用例とは無関係です。

VIEW ファイルの例

以下は、ソース VIEW 記述 custdb を含む VIEW ファイルの例です。

コード リスト 6-1 VIEW **ファイルの例**

```
# BEGINNING OF VIEWFILE
VIEW custdb
#  /* コメント行です。*/
#  /* これもコメント行です。*/
#TYPE        CNAME    FBNAME      COUNT   FLAG  SIZE   NULL
carray       bug      BUG_CURS    4       -     12     "no bugs"
long         custid   CUSTID      2       -      -      -1
short        super    SUPER_NUM   1       -      -      999
long         youid    ID          1       -      -      -1
float        tape     TAPE_SENT   1       -      -      -.001
char         ch       CHR         1       -      -      "0"
string       action   ACTION      4       -     20      "no action"
END
#END OF VIEWFILE
```

フィールド・テーブルの例

以下は、前の節で示した VIEW のコンパイルに必要なフィールド・テーブルの例です。

コード リスト 6-2　**フィールド・テーブルの例**

```
# name          number  type    flags   comments
CUSTID          2048    long    -       -
VERSION_RUN     2055    string  -       -
ID              2056    long    -       -
CHR             2057    char    -       -
TAPE_SENT       2058    float   -       -
SUPER_NUM       2066    short   -       -
ACTION          2074    string  -       -
BUG_CURS        2085    carray  -     -
```

viewc によって生成されるヘッダ・ファイルの例

以下は、VIEW コンパイラによって生成されたヘッダ・ファイルです。viewc に対する入力として、前の節の VIEW ファイルを使用しています。

コード リスト 6-3　viewc **によって生成されるヘッダ・ファイルの例**

```
struct custdb {
char    bug[4][12];             /* null="no bugs"   */
long    custid[2];              /* null=-1          */
short   super;                  /* null=999         */
long    youid;                  /* null=-1          */
float   tape;                   /* null=-0.001000   */
char    ch;                     /* null="0"         */
char    action[4][20];          /* null="no action" */
};
```

mkfldhdr によって生成されるヘッダ・ファイルの例

以下は、mkfldhdr によってフィールド・テーブル・ファイルから生成されたヘッダ・ファイルです。mkfldhdr に対する入力として、前の例のフィールド定義を格納したフィールド・テーブル・ファイルを使用しています。

コード リスト 6-4 mkfldhdr(1) **によって生成されるヘッダ・ファイルの例**

```
/* mkfldhdr を使ってフィールド・テーブルから生成された custdb.flds.h      */
/*       fname       fldid                                               */
/*       -----       -----                                               */
#define  ACTION      ((FLDID)43034)  /* number:2074  type: string */
#define  BUG_CURS    ((FLDID)51237)  /* number:2085  type: carray */
#define  CUSTID      ((FLDID)10240)  /* number:2048  type: long   */
#define  SUPER_NUM   ((FLDID)2066)   /* number:2066  type: short  */
#define  TAPE_SENT   ((FLDID)26634)  /* number:2058  type: float  */
#define  VERSION_RUN ((FLDID)43015)  /* number:2055  type: string */
#define  ID          ((FLDID)10248)  /* number:2056  type: long   */
#define  CHR         ((FLDID)18441)  /* number:2057  type:char   */
```

COBOL COPY ファイルの例

以下は、viewc で -C コマンド行オプションを指定して作成された COBOL COPY ファイル、CUSTDB.cbl です。

コード リスト 6-5 COBOL COPY **ファイルの例**

```
*       VIEWFILE: "t.v"
*       VIEWNAME: "custdb"
            05 BUG OCCURS 4 TIMES              PIC X(12).
*       NULL="no bugs"
            05 CUSTID OCCURS 2 TIMES           PIC S9(9) USAGE IS COMP-5.
*       NULL=-1
            05 SUPER                           PIC S9(4) USAGE IS COMP-5.
```

```
*       NULL=999
             05 FILLER                              PIC X(02).
             05 YOUID                               PIC S9(9) USAGE IS COMP-5.
*       NULL=-1
             05 TAPE                                USAGE IS COMP-1.
*       NULL=-0.001000
             05 CH                                  PIC X(01).
*       NULL='0'
             05 ACTION OCCURS 4 TIMES               PIC X(20).
*       NULL="no action"
              05 FILLER                                 PIC X(03).
```

viewc -C を使用して作成した COBOL COPY ファイルを含む COBOL プログラムの例については、『COBOL を使用した BEA Tuxedo アプリケーションのプログラミング』を参照してください。

VIEWS プログラムの例

以下は、VIEW を使用して構造体とフィールド化バッファをマッピングするプログラムの例です。このプログラムが正しく動作するには、3-1 ページの「FML および VIEWS の環境設定」で説明した環境変数を正しく設定する必要があります。

FML プログラムのコンパイル方法については、『BEA Tuxedo のファイル形式とデータ記述方法』の compilation(5) リファレンス・ページを参照してください。

コード リスト 6-6　VIEWS **プログラムの例**

```
/* VIEW プログラムの例 */
#include stdio.h>
#include "fml.h"
#include "custdb.flds.h"     /* 「viewc によって生成されるヘッダ・ファイル */
/* の例」のフィールド・ヘッダ・ファイル */
#include "custdb.h"          /* 「フィールド・テーブルの例」の viewc で */
/* 作成された C 構造体ヘッダ・ファイル */
#define NF 800
#define NV 400
```

```
extern Ferror;
main()
{
  /* 必要なプログラム変数と FML 関数の宣言 */
    FBFR *fbfr,*Falloc();
    void F_error();
    char *str, *cstruct, buff[100];
    struct custdb cust;

  /* フィールド化バッファの割り当て */
    if ((fbfr = Falloc(NF,NV)) == NULL) {
            F_error("sample.program");
            exit(1);
  }

     /* buff を指すように str ポインタを初期化し、     */
     /* 文字列を buff にコピーし、           */
     /* fbfr バッファのいくつかのフィールドに値を Fadd で追加 */

     str = &buff;
     strcpy(str,"13579");
     if (Fadd(fbfr,ACTION,str,(FLDLEN)6) < 0)
                 F_error("Fadd");
     strcpy(str,"act11");
     if (Fadd(fbfr,ACTION,str,(FLDLEN)6) < 0)
                 F_error("Fadd");
     strcpy(str,"This is a one test.");
     if (Fadd(fbfr,BUG_CURS,str,(FLDLEN)19) < 0)
                 F_error("Fadd");
     strcpy(str,"This is a two test.");
     if (Fadd(fbfr,BUG_CURS,str,(FLDLEN)19) < 0)
                 F_error("Fadd");
     strcpy(str,"This is a three test.");
     if (Fadd(fbfr,BUG_CURS,str,(FLDLEN)21) < 0)
                 F_error("Fadd");

 /* fbfr の現在の内容を出力 */

     printf("fielded buffer before:\n"); Fprint(fbfr);

 /* C 構造体に値を代入 */

    cust.tape = 12345;
    cust.super = 999;
    cust.youid = 80;
    cust.custid[0] = -1; cust.custid[1] = 75;
    str = cust.bug[0][0];
    strncpy(str,"no bugs12345",12);
    str = cust.bug[1][0];
```

```
  strncpy(str,"yesbugs01234",12);
  str = cust.bug[2][0];
  strncpy(str,"no bugsights",12);
  str = cust.bug[3][0];
  strncpy(str,"no bugsysabc",12);
  str = cust.action[0][0];
  strcpy(str,"yesaction");
  str = cust.action[1][0];
  strcpy(str,"no action");
  str = cust.action[2][0];


  strcpy(str,"222action");
  str = cust.action[3][0];
  strcpy(str,"no action");
cust.ch = '0';
cstruct = (char *)&cust;

/* custdb VIEW 記述を使用して、*/
/* C 構造体の値で fbfr バッファを更新 */

if (Fvstof(fbfr,cstruct,FUPDATE,"custdb") < 0) {
           F_error("custdb");
           Ffree(fbfr);
           exit(1);
}

/* 次のコードがデータを */
/* fbfr から cstruct に移す */
/*
if (Fvftos(fbfr,cstruct,"custdb") < 0) {
           F_error("custdb");
           Ffree(fbfr);
           exit(1);
} */

/* C 構造体と */
/* fbfr バッファの値を出力       */

printf("cstruct contains:\en");
printf("action=:%s:\n",cust.action[0][0]);
printf("action=:%s:\n",cust.action[1][0]);
printf("action=:%s:\n",cust.action[2][0]);
printf("action=:%s:\n",cust.action[3][0]);
printf("custid=%ld\n",cust.custid[0]);
printf("custid=%ld\n",cust.custid[1]);
printf("youid=%ld\n",cust.youid);
printf("tape=%f\n",cust.tape);
printf("super=%d\n",cust.super);
printf("bug=:%.12s:\n",cust.bug[0][0]);
```

```
  printf("bug=:%.12s:\n",cust.bug[1][0]);
  printf("bug=:%.12s:\n",cust.bug[2][0]);
  printf("bug=:%.12s:\en",cust.bug[3][0]);
  printf("ch=:%c:\n\n",cust.ch);

 printf("fielded buffer after:\n");
  Fprint(fbfr);
  Ffree(fbfr);
  exit(0);

}
```

bankapp での VIEWS の使用例

bankapp は、BEA Tuxedo システムに同梱されるサンプル・アプリケーションです。このアプリケーション・プログラムには、VIEWS 構造を使用した 2 つのファイルが組み込まれています。サンプル内の構造体は、FML バッファにマッピングされない構造体なので、構造体メンバへのデータの入出力に FML 関数は使用されていません。

$TUXDIR/apps/bankapp/audit.c は、コマンド行オプションを使って、型付きバッファ VIEW にサービス要求を設定する方法を決定するクライアント・プログラムです。

サーバ $TUXDIR/apps/bankapp/BAL.ec のコードは、サービス要求を受け付け、ESQL 文の公式化に使用される VIEW バッファからのフィールドを示します。

関連項目

- 『BEA Tuxedo コマンド・リファレンス』の「viewc、viewc32(1)」
- 『BEA Tuxedo コマンド・リファレンス』の「mkfldhdr、mkfldhdr32(1)」

bankapp での FML の使用例

bankapp は、BEA Tuxedo システムに同梱されるサンプル・アプリケーションです。

```
ACCT.ec
BTADD.ec
TLR.ec
```

上記のサーバでは、FML 型付きバッファ (bankapp クライアントである bankclt からサーバに渡される) のデータを操作するための FML 関数が使用されています。

これらのサーバでは、FML 関数の Falloc、Falloc32(3fml) および Frealloc、Frealloc32(3fml) の代わりに、ATMI 関数の tpalloc(3c) および tprealloc(3c) を使用してメッセージ・バッファを割り当てます。

A FML エラー・メッセージ

次は、FML プログラムの実行時に発生するエラーのエラー・コード、エラー番号、およびエラー・メッセージの一覧です。

表 F-1FML のエラー・コードとエラー・メッセージ

エラー・コード	#	エラー・メッセージ
FALIGN	1	フィールド・バッファが境界付けされていません。
FNOTFLD	2	バッファがフィールド化されていません。
FNOSPACE	3	フィールド・バッファにスペースがありません。
FNOTPRES	4	該当するフィールドがありません。
FBADFLD	5	未知のフィールド番号か、未知のフィールド型が指定されました。
FTYPERR	6	フィールド型が正しくありません。
FEUNIX	7	UNIX システム・コールでエラーが発生しました。
FBADNAME	8	未知のフィールド名が指定されました。
FMALLOC	9	メモリの割り当てに失敗しました。
FSYNTAX	10	ブール式の文法が正しくありません。
FFTOPEN	11	フィールド・テーブルが見つからないか、オープンできません。

表 F-1FML のエラー・コードとエラー・メッセージ (続き)

エラー・コード	#	エラー・メッセージ
FFTSYNTAX	12	フィールド・テーブルに文法エラーがあります。
FEINVAL	13	関数の引き数が無効です。
FBADTBL	14	フィールドテーブルに破壊的な同時アクセスが行われました。
FBADVIEW	15	VIEW 記述が見つからないか、取得できません。
FVFSYNTAX	16	VIEW ファイルに文法エラーがあります。
FVFOPEN	17	VIEW ファイルが見つからないか、オープンできません。
FBADACM	18	ACM に負の値が含まれています。
FNOCNAME	19	cname が見つかりません。
FEBADOP	20	指定されたフィールド・タイプは無効です。