BEA Tuxedo®

Programming a BEA Tuxedo ATMI Application Using FML

Version 10.0
Document Released: September 28, 2007

Contents

3. Setting Up Your Environment for FML and VIEWS

4. Defining and Using Fields

5. Field Manipulation Functions

6. FML and VIEWS Examples

A. FML Error Messages

CHAPTER 1

Introduction to FML Programming

This topic includes the following sections:

What Is FML?

Field Manipulation Language, or FML, is a set of C language functions for defining and manipulating storage structures called fielded buffers, which contain attribute-value pairs in fields. The attribute is the field's identifier, and the associated value represents the field's data content.

Fielded buffers provide an excellent structure for communicating parameterized data between cooperating processes, by providing named access to a set of related fields. Programs that need to communicate with other processes can use the FML software to provide access to fields without concerning themselves with the structures that contain them.

FML also provides a facility called VIEWS that allows you to map fielded buffers to C structures or COBOL records, and vice-versa. The VIEWS facility lets you perform lengthy manipulations of data in structures rather than in fielded buffers; applications run faster if data is transferred to structures for manipulation. Thus the VIEWS facility allows the data independence of fielded buffers to be combined with the efficiency and simplicity of classic record structures.

Two interfaces are available for FML and the VIEWS facility:

- FML and VIEWS accommodate 16-bit field identifiers, field lengths, field occurrences, and record lengths.
- FML32 and VIEW32 accommodate 32-bit field identifiers, field lengths, field occurrences, and record lengths. The type definitions, header files, function names, and command names used in this interface include a “32” suffix.

How Does FML Fit into the BEA Tuxedo System?

Within the BEA Tuxedo system, FML functions are used to manipulate fielded buffers in the context of ATMI applications.

Data entry programs written for the core portion of the BEA Tuxedo system use FML functions; these programs use fielded buffers to forward user data entered at a terminal to other processes. If you write ATMI applications that receive input in fielded buffers from data entry programs, you will need to use FML functions.

Even if you choose to develop your own applications programs for handling user input and output or if programs are written to pass messages between processes, you may still decide to use FML to deal with fielded buffers passed between these programs.

BEA Tuxedo Typed Buffers

Typed buffers is a feature of the BEA Tuxedo system that grew out of the FML idea of a fielded buffer. Two of the standard buffer types delivered with the BEA Tuxedo system are FML typed buffers and VIEW typed buffers. One difference between the two is that BEA Tuxedo VIEW buffers can be totally unrelated to an FML fielded buffer.

In this text we show how a VIEW is a structured version of an FML record. In other documents, such as *Programming a BEA Tuxedo ATMI Application Using C*, we present VIEW as one of several available BEA Tuxedo buffer types.

FML Terminology

Field Identifier

A field identifier (`fldid`) is a tag for an individual data item in an FML record or fielded buffer. The field identifier consists of the name of the field (a number) and the type of data in the field.

Fielded Buffer

A fielded buffer is a data structure in which each data item is accompanied by an identifying tag (a field identifier) that includes the type of the data and a field number.

Field Types

FML fields and fielded buffers are typed. They can be any of the standard C language types: `short`, `long`, `float`, `double`, and `char`. The following types are also supported: `string` (a series of characters ending with a NULL character), `carray` (a character array), `mbstring` (a multibyte character array—available in Tuxedo release 8.1 or later), `ptr` (a pointer to a buffer), `fml32` (an embedded FML32 buffer), and `view32` (an embedded VIEW32 buffer). The `mbstring`, `ptr`, `fml32`, and `view32` types are supported only for the FML32 interface. The corresponding types in COBOL are `COMP-5`, `COMP-1`, `COMP-2`, and `PIC X` with the following exceptions: currently, no corresponding types in COBOL exist for `mbstring`, `ptr`, `fml32`, and `view32`. A C packed decimal type is also supported in VIEWS for integration with `COBOL COMP-3`.

VIEWS

VIEWS is a facility of the Field Manipulation Language that allows the exchange of data between fielded buffers and C structures or COBOL records, by specifying mappings of fields to members of structures/records. If extensive manipulations of fielded buffer information are to be done, transferring the data to structures will improve performance. Information in a fielded buffer can be extracted from the fields in the buffer and placed in a structure using VIEWS functions, manipulated, and the updated values returned to the buffer, again using VIEWS functions. VIEWS can also be used independently of FML, particularly in support of COBOL records.

CHAPTER 2

FML and VIEWS Features

This topic includes the following sections:

Dividing Records into Fields: Data Structures Versus Fielded Buffers

Except under unusual conditions where a data record is a complete and indivisible entity, you need to be able to break records into fields to be able to use or change the information the record contains. In an ATMI environment, records can be divided into fields through either of the following:

- C language data structures or COBOL records
- Fielded buffers

Using Structures to Divide Records into Fields

One common way of subdividing records is with a structure that divides a contiguous area of storage into fields. The fields are given names for identification; the kind of data carried in each field is shown by a data type declaration.

For example, if a data item in a C language program is to contain information about an employee's identification number, name, address, and gender, it could be set up with a structure such as the following:

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

Here the data type of the field named `empid` is declared to be a long integer, `name` and `addr` are declared to be character arrays of 20 and 40 characters respectively, and `gender` is declared to be a single character, presumably with a range of `m` or `f`.

If, in your C program, the variable `p` points to a structure of type struct `S`, the references `p–>empid`, `p–>name`, `p–>addr` and `p–>gender` can be used to address the fields.

The COBOL COPY file for the same data structure would be as follows (the application would supply the 01 line):

```
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).
```

If, in your COBOL program, the 01 line is named `MYREC`, the references `EMPID IN MYREC`, `NAME IN MYREC`, `ADDR IN MYREC`, and `GENDER IN MYREC` can be used to access the fields.

Although this method of representing data is widely used and is often appropriate, it has two major potential disadvantages:

- Any time the data structure is changed, all programs using the structure have to be recompiled.

- The size of the structure and the offsets of the component fields are all fixed, which often results in wasted space, since (a) not all fields always contain a value, and (b) fields tend to be sized to hold the largest likely entry.

Using Fielded Buffers to Divide Records into Fields

Fielded buffers provide an alternative method for subdividing a record into fields.

A fielded buffer is a data structure that provides associative access to the fields of a record; that is, the name of a field is associated with an identifier that includes the storage location as well as the data type of the field.

The main advantage of the fielded buffer is data independence. Fields can be added to the buffer, deleted from it, or changed in length without forcing programs that reference the fields to be recompiled. To achieve this data independence, fields are:

- Referenced by an identifier rather than the fixed offset prescribed by record structures.
- Accessed only through function calls.

Fielded buffers can be used throughout the ATMI environment as the standard method of representing data sent between cooperating processes.

How Fielded Buffers Are Implemented with FML

Fielded buffers are created, updated, accessed, input, and output via Field Manipulation Language (FML). FML provides:

- A convenient and standard discipline for creating and manipulating fielded buffers.
- Data independence to programs that make use of fielded buffers.

FML is implemented as a library of functions and macros that can be called from C programs. It provides a separate set of functions for:

- Creating, updating, accessing, and manipulating fielded buffers.
- Converting data from one type to another upon input to (or output from) a fielded buffer structure.
- Transferring data between fielded buffers and C structures or COBOL records.

The last set of functions listed above constitutes the FML VIEWS software. VIEWS is a set of functions that exchange data between FML fielded buffers and structures in C or COBOL

language application programs. When a program receives a fielded buffer from another process, the program has the choice of:

- Operating on the buffer data directly in the buffer using FML function calls (this is not available in COBOL).
- Transferring the data from the fielded buffer to a structure using VIEWS functions, and then operating on the data in the structure using normal C or COBOL statements.

If you need to perform lengthy manipulations on buffer data, the performance of your program can be improved by transferring fielded buffer data to structures or records, and operating on the data using normal C or COBOL statements. Then you can put the data back into a fielded buffer (again using VIEWS functions), and send the buffer off to another process.

Before you can use VIEWS, you must set up your program such that it can recognize the format of incoming fielded buffer data. You can do this setup task by using a set of view descriptions kept in a cache on your system.

A view description is created and stored in a source viewfile. The view description maps fields in fielded buffers to members in C structures or COBOL records. The source view descriptions are compiled, and can then be used to map data transferred between fielded buffers and C structures or COBOL records in a program.

By keeping view descriptions cached in a central file, you can increase the data independence of your programs; you only need to change the view description(s) and recompile them to effect changes in data format throughout an application that uses VIEWS.

FML Features

This topic includes the following sections:

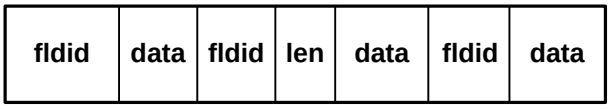
What Is a Fielded Buffer?

A fielded buffer is a data structure that provides associative access to the fields of a record.

Each field in an FML fielded buffer is labeled with an integer that combines information about the data type of the accompanying field with a unique identifying number. The label is called the field identifier, or `fldid`. For variable-length items, the `fldid` is followed by a length indicator.

A buffer can be represented as a sequence of `fldid`/data pairs, with `fldid`/length/data triples for variable-length items, as shown in the following diagram.

Figure 2-1 Fielded Buffer



In the header file that is included (with `#include`) whenever FML functions are used (`fml.h` or `fml32.h`), field identifiers are defined (with `typedef`) as `FLDID` (or `FLDID32` for FML32), field value lengths as `FLDLEN` (`FLDLEN32` for FML32), and field occurrence numbers as `FLDOCC` (`FLDOCC32` for FML32).

Supported Field Types

The supported field types are `short`, `long`, `float`, `double`, `char`, `string`, `carray` (character array), `mbstring` (multibyte character array—available in Tuxedo release 8.1 or later), `ptr` (pointer to a buffer), `fml32` (an embedded FML32 buffer), and `view32` (an embedded VIEW32 buffer). The `mbstring`, `ptr`, `fml32`, and `view32` types are supported only for the FML32 interface. These types are included as `#define` statements in `fml.h` (or `fml32.h`), as shown in the following listing.

Listing 2-1 Definitions of FML Field Types in fml.h and fml32.h

```
#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, and FLD_MBSTRING are all arrays, but differ in the following way:

- A FLD_STRING is a variable-length array of non-NULL characters terminated by a NULL.
- A FLD_CARRAY or FLD_MBSTRING is a variable-length array of bytes, any of which may be NULL.

Functions that add or change a field have a FLDLEN argument that must be filled in when you are dealing with FLD_CARRAY or FLD_MBSTRING fields. The size of a string or carray is limited to 65,535 characters in FML, and 2 billion bytes for FML32.

It is not a good idea to store unsigned data types in fielded buffers. You should either convert all unsigned short data to long or cast the data into the proper unsigned data type whenever you retrieve data from fielded buffers (using the FML conversion functions).

Most FML functions do not perform type checking; they expect that the value you update or retrieve from a fielded buffer matches its native type. For example, if a buffer field is defined to be a FLD_LONG, you should always pass the address of a long value. The FML conversion functions convert data from a user specified type to the native field type (and from the field type to a user specified type) in addition to placing the data in (or retrieving the data from) the fielded buffer.

The FLD_PTR field type makes it possible to embed pointers to application data in an FML32 buffer. Applications can add, change, access, and delete pointers to data buffers. The buffer pointed to by a FLD_PTR field must be allocated using the tpalloc(3c) call. The FLD_PTR field type is supported only in FML32.

The FLD_FML32 field type makes it possible to store an entire record as a single field in an FML32 buffer. Similarly, the FLD_VIEW32 field type allows an entire C structure to be stored as a single field in an FML32 buffer. The FLD_FML32 and FLD_VIEW32 field types are supported only in FML32.

Type int in VIEWS

In addition to the data types supported by most FML functions, VIEWS indirectly supports type `int` in source view descriptions. When the view description is compiled, the view compiler automatically converts any `int` types to either short or long types, depending on your machine. For more information, see "VIEWS Features" on page 2-10.

Type dec_t in VIEWS

VIEWS also supports the `dec_t` packed decimal type in source view descriptions. This data type is useful for transferring VIEW structures to COBOL programs. In a C program using the `dec_t` type, the field must be initialized and accessed using the functions described in `decimal(3c)` in the *BEA Tuxedo ATMI C Function Reference*. Within the COBOL program, the field can be accessed directly using a packed decimal (`COMP-3`) definition. Because FML does not support a `dec_t` field, this field is automatically converted to the data type of the corresponding FML field in the fielded buffer (for example, a string field) when converting from a VIEW to FML.

Field Name-to-Identifier Mappings

In the BEA Tuxedo system, fields are usually referred to by their field identifier (`fldid`), an integer. (Refer to "Defining Field Names and Identifiers" on page 4-2 for a detailed description of field identifiers.) This allows you to reference fields in a program without using the field name, which may change.

Identifiers are assigned (mapped) to field names through one of the following:

- Field table files (which are ordinary UNIX files)
- C language header (`#include`) files

A typical application might use one, or both of the above methods to map field identifiers to field names.

In order for FML to access the data in fielded records, there must be some way for FML to access the field name/identifier mappings. FML gets this information in one of two ways:

- At run time, through UNIX field table files, and FML mapping functions
- At compile time, through C header files

Field name/identifier mapping is not available in COBOL.

Run Time: Field Table Files

Field name/identifier mappings can be made available to FML programs at run time through field table files. It is the responsibility of the programmer to set two environment variables that tell FML where the field name/identifier mapping table files are located.

The environment variable `FLDTBLDIR` contains a list of directories where field tables can be found. The environment variable `FIELDTBLS` contains a list of the files in the table directories that are to be used. For FML32, the environment variable names are `FLDTBLDIR32` and `FIELDTBLS32`.

Within application programs, the FML function `Fldid()` provides for a run-time translation of a field name to its field identifier. `Fname()` translates a field identifier to its field name (see `Fldid(3fml)` and `Fname(3fml)`). (The function names for FML32 are `Fldid32` and `Fname32`.) The first invocation of either function causes space in memory to be dynamically allocated for the field tables and the tables to be loaded into the address space of the process. The space can be recovered when the tables are no longer needed. (Refer to "Loading Field Tables" on page 4-5 for more information.)

This method should be used when field name/identifier mappings are likely to change throughout the life of the application. This topic is covered in more detail in "Defining and Using Fields" on page 4-1.

Compile Time: Header Files

Use `mkfldhdr()` (or `mkfldhdr32()`) to make header files out of field table files. These header files are included (with `#include`) in C programs, and provide another way to map field names to field identifiers: at compile time. For more information on `mkfldhdr, mkfldhdr32(1)`, refer to *BEA Tuxedo Command Reference*.

Using field header files, the C preprocessor converts all field name references to field identifiers at compile time; thus, you do not need to use the `Fldid()` or `Fname()` functions as you would with the field table files described in the previous section.

If you always know the field names needed by your program, you can save some data space by including your field table header files (with `#include`). The space saving allows your program to get to the task at hand more quickly.

Because this method resolves mappings at compile time, however, it should not be used if the field name/identifier mappings in the application are likely to change. For more information, see "Defining and Using Fields" on page 4-1.

Fielded Buffer Indexes

When a fielded buffer has many fields, access is expedited in FML by the use of an internal index. The user is normally unaware of the existence of this index.

Fielded buffer indexes do, however, take up space in memory and on disk. When you store a fielded buffer on disk, or transmit a fielded buffer between processes or between computers, you can save disk space and/or transmittal time by first discarding the index.

The `Funindex()` function enables you to discard the index. When the fielded buffer is read from disk (or received from a sending process), the index can be explicitly reconstructed with the `Findex()` function.

Note that these space savings do not apply to memory. The `Funindex()` function does not recover in-core memory used by the index of a fielded buffer.

For more information, refer to `Funindex, Funindex32(3fml)` or `Findex, Findex32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Multiple Occurrence Fields in a Fielded Buffer

Any field in a fielded buffer can occur more than once. Many FML functions take an argument that specifies which occurrence of a field is to be retrieved or modified. If a field occurs more than once, the first occurrence is numbered 0, and additional occurrences are numbered sequentially. The set of all occurrences makes up a logical sequence, but no overhead is associated with the occurrence number (that is, it is not stored in the fielded buffer).

If another occurrence of a field is added, it is added at the end of the set and is referred to as the next highest occurrence. When an occurrence other than the highest is deleted, all higher occurrences of the field are shifted down by one (for example, occurrence 6 becomes occurrence 5, 5 becomes 4, and so on).

Boolean Expressions and Fielded Buffers

The next action taken by an application program is frequently determined by the value of one or more fields in a fielded buffer received (by the application) from another source, such as a user's terminal or a database record. FML provides several functions that create boolean expressions on fielded buffers or VIEWs and determine whether a given buffer or VIEW meets the criteria specified by the expression.

Once you create a Boolean expression, it is compiled into an evaluation tree. The evaluation tree is then used to determine whether a fielded buffer or VIEW matches the specified Boolean conditions.

For instance, a program may read a data record into a fielded buffer (Buffer A), and apply a Boolean expression to the buffer. If Buffer A meets the conditions specified by the Boolean expression, then an FML function is used to update another buffer, Buffer B, with data from Buffer A.

VIEWS Features

The VIEWS facility is particularly useful when a program does a lot of processing on the data in a fielded buffer, either after the program has received the buffer or before the program sends the buffer to another program.

Under such conditions, you may improve processing efficiency by using VIEWS functions to transfer fielded buffer data from the buffer to a C structure before you manipulate it. Processing efficiency is improved because C functions require less processing time than FML functions for manipulating fields in a buffer. When you finish processing the data in the C structure, you can transfer that data back to the fielded buffer before sending it to another program.

The VIEWS facility has the following features:

- You can create `source view descriptions` that specify C structure-to-fielded buffer mappings or COBOL record-to-fielded buffer mappings, and make possible the transfer of data between structures and buffers.
- The `viewc` or `viewc32` view compiler is used to generate `object view descriptions` (stored in binary files) that are interpreted by your application programs at run time. The compiler also generates header files that can be used in C programs to define the structures used in view descriptions, and optionally generates COPY files that can be used in COBOL programs to define the records used in the view descriptions. For more information about these view compilers, see `viewc, viewc32(1)` in *BEA Tuxedo Command Reference*.
- A view disassembler is provided to translate object view descriptions into readable form (that is, back into source view descriptions). The output of the disassembler can be re-input to the view compiler.
- Data transfers from C structures or COBOL records to fielded buffers can be done in any one of four modes: `FUPDATE`, `FJOIN`, `FOJOIN`, or `FCONCAT`. These modes are similar to the ones supported by the following FML functions: `Fupdate, Fupdate32(3fml)`,

Fjoin, Fjoin32(3fml), Fojoin, Fojoin32(3fml), and Fconcat, Fconcat32(3fml).

- At run time object view descriptions are read into a viewfile cache on demand, and remain there until the cache is full. When the cache is full and an object view description that is not in the cache is needed, the least recently accessed object view description is removed from the cache to make room for the new one.
- All types supported by FML can be used in view descriptions with the exception of FLD_PTR, FLD_FML32, and FLD_VIEW32. In addition, integer and packed decimal are supported.
- When transferring data between fielded buffers and structures, the source data is automatically converted to the type of the destination data; for instance, if a string field is mapped to an integer member, the string is converted to an integer using Ftypcvt() automatically. For more information, refer to Ftypcvt, Ftypcvt32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.
- Multiple field occurrences are supported.
- User-specified and default NULL values in view descriptions are supported.
- Functions are available for compiling and evaluating Boolean expressions against application data in a VIEW.

A source viewfile is an ordinary text file that contains one or more source view descriptions. Source viewfiles are used as input to a view compiler—viewc or viewc32—which compiles the source view descriptions and stores them in object viewfiles. For more information on the view compiler, refer to viewc, viewc32(1) in *BEA Tuxedo Command Reference*.

The view compiler also creates C header files for object viewfiles. These header files can be included in application programs to define the structures used in object view descriptions.

The view compiler optionally creates COBOL COPY files for object viewfiles. These COPY files can be included in COPY programs to define the record formats used in object view descriptions.

NULL values are used to indicate empty members in a structure, and can be specified by the user for each structure member in a viewfile. If the user does not specify a NULL value for a member, default NULL values are used.

Note that a structure member containing the NULL value for that member is not transferred during a structure-to-fielded buffer transfer.

It is also possible to inhibit the transfer of data between a C or COBOL structure member and a field in a fielded buffer, even though a mapping exists between them. This is specified in the source viewfile.

The FML VIEWS functions are `Fvstof()`, `Fvftos()`, `Fvnull()`, `Fvopt()`, `Fvselinit()`, and `Fvsinit()`. For COBOL, the VIEWS facility provides two procedures: `FVSTOF` and `FVFTOS`. Upon calling any view function, the named object viewfile, if found, is loaded into the viewfile cache automatically. Each file specified in the environment variable `VIEWFILES` is searched in order (see "Setting Up Your Environment for FML and VIEWS" on page 3-1). The first object viewfile with the specified name is loaded. Subsequent object viewfiles with the same name, if any, are ignored. For more information on the FML VIEWS functions, refer to *BEA Tuxedo ATMI FML Function Reference*.

Note that arrays of structures, pointers, unions, and typedefs are not supported in VIEWS.

Multiple Occurrence Fields in VIEWS

Because VIEWS is concerned with moving fields between fielded buffers and C structures or COBOL records, it must deal with the possibility of multiple occurrence fields in the buffer.

To store multiple occurrences of a field in a structure, a member is declared as an array in C or with the OCCURS clause in COBOL; each occurrence of a field occupies one element of the array. The size of the array reflects the maximum number of field occurrences in the buffer.

When transferring data from fielded buffers to C structures or COBOL records, if the number of elements in the receiving array is greater than the number of occurrences in the fielded buffer, the extra elements are assigned the (default or user-specified) NULL value. If the number of occurrences in the buffer is greater than the number of elements in the array, the extra occurrences in the buffer are ignored.

When data is transferred from C structures or COBOL records to fielded buffers, array members with the value equal to the (default or user-specified) NULL values are ignored.

Error Handling for FML Functions

When an FML function detects an error, one of the following values is returned:

- NULL is returned for functions that return a pointer.
- `BADFLDID` is returned for functions that return a `FLDID`.
- `-1` is returned for all others.

All FML function call returns should be checked against the appropriate value above to detect errors.

In all error cases, the external integer `Ferror` is set to the error number as defined in `fml.h`. `Ferror32` is set to the error number for FML32 as defined in `fml32.h`.

The `F_error()` (or `F_error32()`) function is provided to produce a message on the standard error output. It takes one parameter, a string. It prints the argument string, appended with a colon and a blank, and then prints an error message, followed by a newline character. The error message displayed is the one defined for the error number currently in `Ferror`, which is set when errors occur.

To be of most use, the argument string to the `F_error()` (or `F_error32()`) function should include the name of the program that incurred the error. Refer to `F_error, F_error32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

`Fstrerror, Fstrerror32(3fml)` can be used to retrieve the text of an error message from a message catalog; it returns a pointer that can be used as an argument to `userlog(3c)`, or to `F_error()` or `F_error32()`.

For a description of the error codes produced by an FML function, see the entry for that function in *BEA Tuxedo ATMI FML Function Reference*.

CHAPTER 3

Setting Up Your Environment for FML and VIEWS

This topic includes the following sections:

Environment Requirements for FML and VIEWS

Before you can begin to work with FML fielded buffers, or to use the VIEWS functions that move fields between structures and fielded buffers, you must set up your environment to accommodate these methods by setting the necessary environment variables. This section provides instructions for doing so.

FML Directory Structure

The FML software delivered with the BEA Tuxedo system resides in a subtree of the local file system. Several FML modules depend on the subtree structure described here. We assume that you have set the `TUXDIR` environment variable to the full path name of the directory in which the BEA Tuxedo ATMI Server is installed.

The BEA Tuxedo installation directory contains the following subdirectories:

- `include`—contains header files needed by writers of C application code.

- `cobinclude`—contains COPY files needed by writers of COBOL application code. (This directory is named `cobinclu` for operating systems with an 8.3 file name limitation.)
- `bin`—contains the executable commands of FML.
- `lib`—contains subroutine packages of FML. When compiling a program that uses FML functions, you should include `$TUXDIR/lib/libfml.`*`suffix`* and `$TUXDIR/lib/libgp.`*`suffix`* on the C compiler command line to resolve external references. `libfml32.`*`suffix`* contains the FML32 and VIEW32 functions. (The suffix is `.a` for POSIX operating systems without shared objects, `.so.`*`release`* for use of shared objects, `.lib` for Windows; it is part of the BEA Tuxedo system DLL for platforms that use dynamic link libraries.)

C applications in which FML is used must include the following header files in the order shown:

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

The file `fml.h` or `fml32.h` contains definitions for structures, symbolic constants, and macros used by the FML software.

Environment Variables Used by FML and VIEWS

Several environment variables are used by FML and VIEWS.

- The following variable is used in FML to search for system-supplied files:
  - `TUXDIR`—this variable should be set to the topmost node of the installed BEA Tuxedo system software including FML.
- The following variables are used throughout FML to access field table files:
  - `FIELDTBLS`—this variable should contain a comma-separated list of field table files for the application. Files given as full path names are used as is; files listed as relative path names are searched for through the list of directories specified by the `FLDTBLDIR` variable. `FIELDTBLS32` is used for `FML32`. If `FIELDTBLS` is not set, then the single file name `fld.tbl` is used. (`FLDTBLDIR` still applies; see below.)
  - `FLDTBLDIR`—this variable specifies a colon-separated list of directories to be used to find field table files with relative filenames. Its usage is similar to the `PATH` environment variable. If `FLDTBLDIR` is not set or is NULL, then its value is taken to be the current directory. `FLDTBLDIR32` is used for `FML32`.

  For details, see “Defining and Using Fields” on page 4-1.

- VIEWS functions use the same environment variables used by FML (namely, FLDTBLDIR and FIELDTBLS) plus two other environment variables:
  - VIEWFILES—this variable should contain a comma-separated list of object viewfiles for the application. Files given as full path names are used as is; files listed as relative path names are searched for through the list of directories specified by the VIEWDIR variable (see the following list item). VIEWFILES32 is used for VIEW32.
  - VIEWDIR—this variable specifies a colon-separated list of directories to be used to find view object files with relative filenames. It is set and used in the same way that the PATH environment variable is set and used. If VIEWDIR is not set or is NULL, then its value is assumed to be the current directory. VIEWDIR32 is used for VIEW32.
- The following variables are used in FML32 to support the FLD_MBSTRING field type:
  - TPMBENC—this variable specifies the code-set encoding name that the application server or client running BEA Tuxedo 8.1 or later includes for an FLD_MBSTRING field in an FML32 typed buffer. When an application server or client process allocates and sends an FML32 buffer containing a FLD_MBSTRING field, the code-set encoding name defined in TPMBENC is automatically used by Fmbpack32() if its *enc* argument is not defined and its *flag* argument is not set to FBUFENC.

    When the application server or client process receives an FML32 buffer that includes an FLD_MBSTRING field, and assuming another environment variable named TPMBACONV is set, the code-set encoding name defined in TPMBENC is automatically compared to the code-set encoding name included for the FLD_MBSTRING field in the received buffer; if the names are *not* the same, the FLD_MBSTRING field data is automatically converted to the encoding defined in TPMBENC before being delivered to the server or client process.

    TPMBENC has no default value. For an application server or client using FLD_MBSTRING fields, TPMBENC must be defined for automatic conversion to work.

    Note: TPMBENC is used in a similar way for MBSTRING typed buffers.
  - TPMBACONV—this variable specifies whether the application server or client running BEA Tuxedo 8.1 or later automatically converts the FLD_MBSTRING field data in a received FML32 buffer to the encoding defined in TPMBENC. By default, the automatic conversion is turned off, meaning that the FLD_MBSTRING field data is delivered to the destination server or client process *as is*—no encoding conversion. Setting TPMBACONV to any non-NULL value, say Y (yes), turns on the automatic conversion.

    Note: TPMBACONV is used in a similar way for MBSTRING typed buffers.

  For details, see "Converting FLD_MBSTRING Fields" on page 5-54.

VIEW32 Support for MBSTRING

Starting with Tuxedo 9.0, VIEW32 supports MBSTRING typed buffers which correspond to the `FLD_MBSTRING` field type in FML32.

`Fmbpack32(3fml)`prepares an MBSTRING field in a VIEW32 buffer for encoding and `Fmbunpack32(3fml)`extracts it. `TPMBENC` and `TPMBACONV` environment variables are also used in VIEW32.

CHAPTER 4

Defining and Using Fields

This topic includes the following sections:

Preparing to Use FML and VIEWS

Before you can begin to work with FML fielded buffers, or to use the VIEWS functions that move fields between structures and fielded buffers, you must:

- Define fields.
- Make field definitions available to application programs (through field table files and mapping functions at run time, or through C header files at compile time).
- Compile source view descriptions into object view descriptions, and generate corresponding C header files and COBOL COPY files.

These tasks and related activities are described here.

Defining Fields for FML and VIEWS

This topic includes the following sections:

- Creating Field Table Files
- Mapping Field Names to Field IDs
- Loading Field Tables
- Converting Field Tables to Header Files

Defining Field Names and Identifiers

A field identifier (`fieldid`) is defined (with `typedef`) as a `FLDID` (`FLDID32` for FML32), and is composed of two parts: a field type and a field number. The number uniquely identifies the field.

A field number must fall in one of the following ranges:

- For FML: between 1 and 8191, inclusive
- For FML32: between 1 and 33,554,431, inclusive

Field number 0 and the corresponding field identifier 0 are reserved to indicate a bad field identifier (`BADFLDID`). When FML is used with other software that also uses fields, additional restrictions may be imposed on field numbers.

The BEA Tuxedo system conforms to the following conventions for field numbers.

FML Field Numbers		FML32 Field Numbers	
Reserved	**Available**	**Reserved**	**Available**
1-100	101-8191	1-10,000, 30,000,001-33,554,431	10,001-30,000,000

Applications should avoid using the reserved field numbers, although the BEA Tuxedo system does not strictly enforce applications from using them.

The mappings between field identifiers and field names are contained in either field table files or field header files. If you are using field table files you must convert field name references in C programs with the mapping functions described later in this section. Field header files allow the C preprocessor (`cpp`(1) in UNIX reference manuals) to resolve name-to-field ID mappings when a program is compiled.

The functions and programs that access field tables use the environment variables `FLDTBLDIR` and `FIELDTBLS` to specify the source directories and field table files, respectively, that are to be

used. (FLDTBLDIR32 and FIELDTBLS32 are used for the same purpose with FML32.) You should set these environment variables as described in "Setting Up Your Environment for FML and VIEWS" on page 3-1.

The use of multiple field tables allows you to establish separate directories and/or files for separate groups of fields. Note that field names and field numbers should be unique across all field tables, since such tables are capable of being converted into C header files, and field numbers that occur more than once may cause unpredictable results.

Creating Field Table Files

Field table files are created using a standard text editor, such as vi. They have the following format:

- Blank lines and lines beginning with # are ignored.
- Lines beginning with a dollar sign ($) are ignored by the mapping functions but are passed through (without the $) to header files generated by mkfldhdr. (Refer to mkfldhdr, mkfldhdr32(1) in *BEA Tuxedo Command Reference*.) The ability to have lines ignored by the mapping functions is useful, for example, when an application passes C comments, what strings, and so on, to the generated C header file.

  Note: In COBOL applications, however, such lines are not passed through to the COBOL copy files.

- Lines beginning with the string *base contain a base for offsetting subsequent field numbers. This optional feature provides an easy way to group and renumber sets of related fields.
- All other lines should have the form:

  name rel-number type flag comment

  where:

  – *name* is the identifier for the field. It should not exceed the C preprocessor identifier restrictions (that is, it should contain only alphanumeric characters and the underscore character). Internally, the name is truncated to 30 characters, so names must be unique within the first 30 characters.

  – *rel-number* is the relative numeric value of the field. It is added to the current base, if *base is specified, to obtain the field number of the field.

  – *type* is the type of the field. It is specified as one of the following: short, long, float, double, char, string, carray, mbstring, ptr, fml32, or view32.

– The *flag* field is reserved for future use; use a dash (-) in this field.

– *comment* is an optional field that can be used for clarifying information.

Note that these entries must be separated by white space (blanks or tabs).

Field Table Example

The following is an example field table in which the base shifts from 500 to 700. The first field in each group will be numbered 501 and 701, respectively.

Listing 4-1 System Field Table File

```
# 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
```

Mapping Field Names to Field IDs

Run-time mapping is done by the `Fldid()` and `Fname()` functions, which consult the set of field table files specified by the `FLDTBLDIR` and `FIELDTBLS` environment variables. (If FML32 is being used, the `Fldid32()` and `Fname32()` functions reference the `FLDTBLDIR32` and `FIELDTBLS32` environment variables.)

`Fldid` maps its argument, a field name, to a `fieldid`, as shown in the following code:

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

```
...
id = Fldid(name);
```

`Fname` does the reverse translation by mapping its argument, a `fieldid`, to a field name, as shown in the following code:

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

Identifier-to-name mapping is rarely used; it is rare that one has a field identifier and wants to know the corresponding name. One situation in which the field identifier-to-field name mapping can be used is in a buffer print routine designed to display, in an intelligible form, the contents of a fielded buffer.

See Also

- `Fldid, Fldid32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*
- `Fname, Fname32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*

Loading Field Tables

Upon the first call, `Fldid()` loads the field table files and performs the required search. Thereafter, the files are kept loaded. `Fldid()` returns the field identifier corresponding to its argument on success, and returns `BADFLDID` on failure, with `Ferror` set to `FBADNAME`. (If FML32 is being used, `Ferror32` is set, instead.)

To recover the data space used by the field tables loaded by `Fldid()`, you may unload all of the files by calling the `Fnmid_unload()` function.

The function `Fname()` acts in a fashion similar to `Fldid()`, but it provides a mapping from a field identifier to a field name. It uses the same environment variable scheme for determining the field tables to be loaded, but constructs a separate set of mapping tables. On success, `Fname()` returns a pointer to a character string containing the name corresponding to the `fldid` argument. On failure, `Fname()` returns NULL.

Note: The pointer is valid only as long as the table remains loaded.

As with `Fldid()`, failure includes either the inability to find or open a field table (`FFTOPEN`), bad field table syntax (`FFTSYNTAX`), or a no-hit condition within the field tables (`FBADFLD`). The table space used by the mapping tables created by `Fname()` may be recovered by a call to the `Fidnm_unload()` function.

Both mapping functions and other FML functions that use run-time mapping require FIELDTBLS and FLDTBLDIR to be set properly. Otherwise, defaults are used. (For the default values of these environment variables, see "Setting Up Your Environment for FML and VIEWS" on page 3-1.)

See Also

- Fldid, Fldid32(3fml) in *BEA Tuxedo ATMI FML Function Reference*
- Fnmid_unload, Fnmid_unload32(3fml) in *BEA Tuxedo ATMI FML Function Reference*
- Fname, Fname32(3fml) in *BEA Tuxedo ATMI FML Function Reference*
- Fidnm_unload, Fidnm_unload32(3fml) in *BEA Tuxedo ATMI FML Function Reference*

Converting Field Tables to Header Files

The mkfldhdr (or mkfldhdr32) command converts a field table, as described earlier, into a header file suitable for processing by the C compiler. Each line of the generated header file is of the following form:

```
#define fname fieldid
```

where *fname* is the name of the field, and *fieldid* is its field-ID. The field-ID has both the field type and field number encoded in it. The field number is an absolute number, that is, base plus rel-number. The resulting file is suitable for inclusion in a C program.

It is not necessary to use the header file if the run-time mapping functions are used as described in "Mapping Fields to C Structures and COBOL Records" on page 4-8.

The advantage of compile-time mapping of names to identifiers is speed and a decrease of data space requirements. The disadvantage is that changes made to field name/identifier mappings after, for instance, a service routine has been compiled, are not propagated to the service routine. (Under such circumstances, the service routine uses the mappings it has already compiled.)

mkfldhdr translates each field table specified in the FIELDTBLS environment variable to a corresponding header file, the name of which is formed by adding a .h suffix to the field table name. The resulting files are created, by default, in the current directory. If you want your header files to be created in another directory, you may specify that directory with the -d option on the mkfldhdr command line. For more information, refer to mkfldhdr, mkfldhdr32(1) in *BEA Tuxedo Command Reference*.

Examples of Converting Field Tables to Header Files

Examples 1 and 2 show how to set your environment variables and run the `mkfldhdr(1)` command so that three field table files—`${FLDTBLDIR}/maskftbl`, `${FLDTBLDIR}/DBftbl`, and `${FLDTBLDIR}/miscftbl`—are processed, and three include files—`maskftbl.h`, `DBftbl.h` and `miscftbl.h`—are generated in the current directory. For more information, refer to `mkfldhdr, mkfldhdr32(1)` in *BEA Tuxedo Command Reference*.

Example 1

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

Example 2

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

Example 3

Example 3 is the same as Example 1 with one exception: the output files—`maskftbl.h`, `DBftbl.h` and `miscftbl.h`—are generated in the directory indicated by `${FLDTBLDIR}`.

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

```
mkfldhdr -d${FLDTBLDIR}
```

Overriding Environment Variables to Run mkfldhdr

You may override the environment variables (or avoid setting them) when using `mkfldhdr` by specifying, on the command line, the names of the field tables to be converted.

This method does not apply to run-time mapping functions, however. When run-time mapping functions are being used, `FLDTBLDIR` is assumed to be the current directory and `FIELDTBLS` is

assumed to be the list of parameters that the user specified on the command line. For example, the command:

```
mkfldhdr myfields
```

converts the field table file called `myfields` to a field header file called `myfields.h`, and puts the new file in the current directory.

For more information, refer to `mkfldhdr, mkfldhdr32(1)` in *BEA Tuxedo Command Reference*.

Mapping Fields to C Structures and COBOL Records

This topic includes the following sections:

- What Is the VIEWS Facility?
- Creating Viewfiles
- Creating View Descriptions
- Compiling Viewfiles
- Using Header Files Compiled with viewc
- Using COBOL COPY Files Created by the View Compiler
- Displaying Viewfile Information After Compilation

What Is the VIEWS Facility?

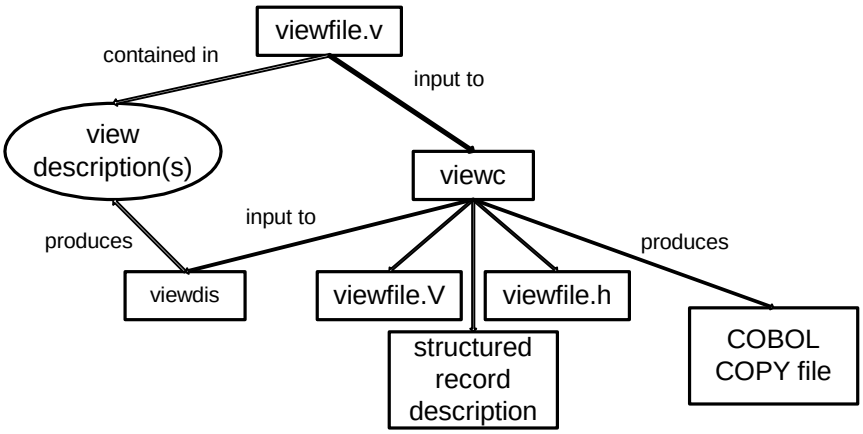
FML VIEWS is a mechanism that enables the exchange of data between fielded buffers and C structures or COBOL records. This facility is provided because it is usually more efficient to perform lengthy manipulations on C structures with C functions than on fielded buffers with FML functions. VIEWS also provides a way for a COBOL program to send and receive messages with a C program that handles FML fielded records.

This section explains how to use VIEWS to provide fielded buffer/structure mappings.

Structure of VIEWS

The following diagram shows the various components of VIEWS and how they relate to one another.

Figure 4-1 Components of the VIEWS Facility



Creating Viewfiles

Source viewfiles are standard text files (created through any standard text editor, such as `vi`) that contain one or more source view descriptions (the actual field-to-structure mappings).

The view compiler produces (among other things) object viewfiles containing the compiled object view descriptions. These object viewfiles can be used, in turn, as input to the view disassembler (`viewdis` or `viewdis32`), which translates the object view descriptions back into their source format (for verification or editing). For more information, refer to `viewdis, viewdis32(1)` in *BEA Tuxedo Command Reference*.

You can create and edit source view descriptions, and edit the output of `viewdis`. You cannot read compiled view descriptions (which are in binary format) directly.

Besides view descriptions, viewfiles may contain comment lines, beginning with # or $. Blank lines and lines beginning with # are ignored by the view compiler, while lines beginning with $ are passed by the view compiler to any header files generated. This convention lets you pass C comments, `what` strings, and so on, to C header files produced by the view compiler.

Note: This convention is not observed for COBOL; lines beginning with $ are not passed through to the COBOL copy files.

Creating View Descriptions

Each source view description in a source viewfile consists of three parts:

- A line beginning with the keyword `VIEW` (never with a `32` suffix), followed by the name of the view description. This name may be composed of alphanumeric characters, including an underscore. Although `viewc` accepts names of up to 33 characters, the practical limit in most cases is 16 characters, since this is the maximum length for a subtype accepted by `tpalloc(3c)`.
- A list of member descriptions.
- A line beginning with the keyword `END`.

The first line of each view description must begin with the keyword `VIEW`, followed by the name of the view description. A member description (or mapping entry) is a line with information about a member in the C structure or COBOL record. A line with the keyword `END` must be the last line in a view description.

The following listing shows the general structure of a source view description.

Listing 4-2 Source View Description

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

In the previous listing:

- *vname* is the name of the view description, and should be a valid C identifier name, since it is also used as the name of a C structure. Underscores are mapped automatically to dashes in the COBOL COPY file.
- *type* is the type of the member, and is specified as one of the following: `int`, `short`, `long`, `char`, `float`, `double`, `string`, `carray`, or `dec_t`. If the value of *type* is "`-`", the default—the value of *fbname*—is used.
- *cname* is the identifier for the structure member, and should be a valid C identifier name, since it is the name of a C structure member. Internally, the *cname* is truncated to 30

characters, so *cnames* must be unique within the first 30 characters. Underscores are mapped automatically to dashes in the COBOL COPY file.

- *fbname* is the name of the field in the fielded buffer. This name must appear in a field table file.
- *count* is the number of elements to be allocated (that is, the maximum number of occurrences to be stored for this member). The value of *count* must be less than or equal to 65,535 for FML, and less than or equal to 2,147,483,647 for FML32.
- *flag* is a comma-separated list of options or "-" (which means that no options are set). For details, see "Specifying flag Options in a View Description" on page 4-11.
- *size* is the size of the member if the type is `string`, `carray`, or `dec_t`. For other types, "-" should be specified; the view compiler computes the size.
  - For `string` or `carray`, the value of *size* must be less than or equal to 65,535 for FML and less than or equal to 2,147,483,647 for FML32.
  - For the `dec_t` type, the value of *size* must be two numbers separated by a comma. The first number represents the number of bytes in the decimal value; it must be greater than 0 and less than 10. The second number represents the number of decimal places to the right of the decimal point; it must be greater than 0 and less than twice the number of bytes minus one.
- `null` is the user-specified NULL value or "-" to indicate the default NULL value for that field. For details, see "Using NULL Values in VIEWS" on page 4-14.

Specifying flag Options in a View Description

The following options can be specified as the `flag` element of a member description in a view description.

`C`

This option requests the generation of a structure member called the associated count member (ACM), in addition to the structure member described in the member description.

When data is being transferred from a fielded buffer to a structure, each ACM in the structure is set to the number of occurrences transferred to the associated structure member.

- A value of 0 in an ACM indicates that no fields were transferred to the associated structure member

– A positive value indicates the number of fields actually transferred to the structure member array.

– A negative value indicates that there were more fields in the buffer than could be transferred to the structure member array. (The absolute value of the ACM equals the number of fields not transferred to the structure).

During a transfer of data from a structure member array to a fielded buffer, the ACM is used to indicate the number of array elements that should be transferred. For example, if the ACM of a member is set to N, the first N non-NULL fields are transferred to the fielded buffer. If N is greater than the dimension of the array, it defaults to the dimension of the array. In either event, after the transfer takes place, the ACM is set to the actual number of array members transferred to the fielded buffer.

The type of an ACM in the C header file is declared to be `short` for FML and `long` for FML32, and its name is generated as `C_`*`cname`*, where *`cname`* is the `cname` entry for which the ACM is declared. For example, an ACM for a member named `parts` is declared as follows:

```
short C_parts;
```

For a COBOL COPY file, the name is generated as `C-`*`cname`* and the type is declared as follows:

– For FML: `PIC S9(4) USAGE COMP-5`

– For FML32: `PIC S9(9) USAGE COMP-5`

Note: It is possible for the generated ACM name to conflict with structure members with names that begin with a `C_` prefix. Such conflicts are reported by the view compiler, and are considered fatal errors by the compiler. For example, the name `C_parts` for a structure member conflicts with the name of an ACM generated for the member `parts`.

`F`

Specifies one-way mapping from structure or record to fielded buffer. The mapping of a member with this option is effective only when transferring data from structures to fielded buffers. This option is ignored if the `-n` command-line option is specified.

`L`

This option is used only for member descriptions of type `carray` or `string` to indicate the number of bytes transferred for these possibly variable length fields. If a `carray` or `string` field is always used as a fixed length data item, then this option provides no benefit.

The L option generates an associated length member (ALM) for a structure member of type carray or string. When transferring data from a fielded buffer to a structure, the ALM is set to the length of the corresponding transferred fields. If the length of a field in the fielded buffer exceeds the space allocated in the mapped structure member, only the allocated number of bytes is transferred. The corresponding ALM is set to the size of the fielded buffer item. Therefore, if the ALM is greater than the dimension of the structure member array, the fielded buffer information is truncated on transfer.

When data is being transferred from a structure member to a field in a fielded buffer, the ALM is used to indicate the number of bytes to transfer to the fielded buffer, if it is a carray type field. For strings, the ALM is ignored on transfer, but is set afterwards to the number of bytes transferred. Note that because carray field may be of zero length, an ALM of 0 indicates that a zero-length field should be transferred to the fielded buffer, unless the value in the associated structure member is the NULL value.

An ALM is defined in the C header file as an unsigned short for FML and an unsigned long for FML32, and has a generated name of L_*cname*, where *cname* is the name of the structure for which the ALM is declared.

If the number of occurrences of the member for which the ALM is declared is 1 (or defaults to 1), then the ALM is declared as:

```
unsigned short L_cname;
```

whereas if the number of occurrences is greater than 1, say N, the ALM is declared as:

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

and is referred to as an ALM Array. In this case, each element in the ALM array refers to a corresponding occurrence of the structure member (or field). For the COBOL COPY file, the type is declared to be PIC 9(4) USAGE COMP-5 for FML and PIC 9(9) USAGE COMP-5 for FML32, and its name is generated as L-*cname*. The COBOL OCCURS clause is used to define multiple occurrences if the member occurs multiple times.

Note: It is possible for the generated ALM name to conflict with structure members with names that begin with an L_ prefix. Such conflicts are reported by the view compiler, and are considered fatal errors by the compiler. For example, the name L_parts for a structure member conflicts with the name of an ALM generated for the member parts.

N

Specifies zero-way mapping; no fielded buffer is mapped to the structure. This option can be used to allocate fillers in C structures or COBOL records. It is ignored if the -n command-line option is specified.

P

This option can be used to affect what VIEWS interprets as a NULL value for `string` and `carray` type structure members. If this option is not used, a structure member is NULL if its value is equal to the user-specified NULL value (without considering any trailing NULL characters).

If this option is set, however, a member is NULL if its value is equal to the user-specified NULL value with the last character propagated to full length (without considering any trailing NULL character).

A member whose value is NULL is not transferred to the destination buffer when data is transferred from the C structure or COBOL record to the fielded buffer. For example, a structure member `TEST` is of type `carray[25]` and a user-specified NULL value "`abcde`" is established for it. If the `P` option is not set, `TEST` is considered NULL if the first five characters are `a`, `b`, `c`, `d`, and `e`, respectively. If the `P` option is set, `TEST` is NULL if the first four characters are `a`, `b`, `c`, and `d`, respectively, and the rest of the `carray` contains the character "`e`" (that is, 21 `e`'s).

This option is ignored if the `-n` command-line option is specified.

S

Specifies one-way mapping from fielded buffer to structure or record. The mapping of a member with this option is effective only when transferring data from fielded buffers to structures. This option is ignored if the `-n` command line option is specified.

Using NULL Values in VIEWS

NULL values are used in VIEWS to indicate empty C structure or COBOL record members. Default NULL values are provided; you may also define your own.

The default NULL value for all numeric types is `0` (`0.0` for `dec_t`); for `char` types, it is "`\0`"; and for `string` and `carray` types, it is " ".

Escape convention constants can also be used to specify a NULL value. The view compiler recognizes the following escape constants: `\`*`ddd`* (where *d* is an octal digit), `\0`, `\n`, `\t`, `\v`, `\b`, `\r`, `\f`, `\\`, `\'`, and `\"`.

`String`, `carray`, and `char` NULL values may be enclosed in double or single quotes. Unescaped quotes within a user-defined NULL value are not accepted by the view compiler.

Alternatively, an element is NULL if its value is the same as the NULL value for that element, except in the following cases:

- If the P option is set for the structure member, and the structure member is of string or carray type; see the preceding section for details on the P option flag.
- If a member is of type string, its value must be the same string as the NULL value.
- If a member is of type carray and the NULL value is of length N, then the first N characters in the carray must be the same as the NULL value.

You can also specify the keyword "NONE" in the NULL field of a view member description, which means there is no NULL value for the member.

The maximum size of default values for string and character array (carray) members is 2660 characters.

Note: Note that for string members, which usually end with a "\0", a "\0" is not required as the last character of a user-defined NULL value.

Compiling Viewfiles

viewc is a view compiler program for FML and viewc32 is used for FML32. It takes a source viewfile and produces an object viewfile, which is interpreted at run time to effect the actual mapping of data. At run time, a C compiler must be available for viewc. The command line looks like the following:

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

where *viewfile* is the name of a source viewfile containing source view descriptions. You may specify one or more *viewfiles* on the command line.

If the -C option is specified, then one COBOL COPY file is created for each VIEW defined in the *viewfile*. These copy files are created in the current directory.

The -n option can be used when compiling a view description file for a C structure or COBOL record that does not map to an FML buffer.

By default, all views in *viewfile* are compiled and two or more files are created: an object viewfile (suffixed with ".V"), and a header file (suffixed with ".h") for each viewfile. For an illustration of the VIEWS components, see the diagram titled "Components of the VIEWS Facility" on page 4-9.

The name of the object viewfile is *viewfile*.V. It is created in the current directory. The -d option can be used to specify an alternate directory. Header files are created in the current directory.

Note: For those operating systems that are not case-sensitive, such as Windows, the object viewfile is given a .vv suffix.

For more information, refer to viewc, viewc32(1) in *BEA Tuxedo Command Reference*.

Using Header Files Compiled with viewc

You can use header files created by the view compiler (viewc) in any C application programs to declare a C structure described by views. For example, the following view description:

```
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
```

produces a C header file that looks like this:

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

For more information, refer to viewc, viewc32(1) in *BEA Tuxedo Command Reference*.

Using COBOL COPY Files Created by the View Compiler

COBOL COPY files created by the view compiler with the -C option can be used in any COBOL application programs to declare COBOL records described by views. For example, the COBOL COPY file for the previous view description looks like the following in the file TEST.cbl:

```
*       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).
```

Note that the COPY filename is automatically converted to uppercase by the view compiler. The COPY file is included in a COBOL program as follows:

```
01 MYREC COPY TEST.
```

For a more complete description of the output in the resulting COPY files, see *Programming a BEA Tuxedo ATMI Application Using COBOL*.

Displaying Viewfile Information After Compilation

The view disassembler, `viewdis`, disassembles an object viewfile produced by the view compiler and displays view information in source viewfile format. In addition, it displays the offsets of structure members in the associated structure.

The ability to view the information in this type of format is useful for verifying that an object view description is correct.

To run the view disassembler, enter the following command:

`viewdis` *`objviewfile . . .`*

By default, *`objviewfile`* in the current directory is disassembled. If this file is not found in the current directory, an error message is displayed. You can specify one or more view object files on the command line.

The output of `viewdis` looks similar to the original source view description. It can be edited and re-input to `viewc`. The order of the lines in the output of `viewdis` may be different from the order of the lines in the original source view description, but this difference is irrelevant in determining whether the object file is correct.

For more information, refer to `viewdis, viewdis32(1)` in *BEA Tuxedo Command Reference*.

CHAPTER 5

Field Manipulation Functions

This topic includes the following sections:

- Boolean Functions
- VIEW Conversion to and from Target Format

About This Section

This section describes all FML and VIEWS functions except the run-time mapping functions described in "Defining and Using Fields" on page 4-1.

FML functions are not directly available for COBOL programs. A procedure called `FINIT` is available to initialize a record for receiving FML data, and the `FVSTOF` and `FVFTOS` procedures are available to convert a COBOL record into an FML buffer, and vice-versa. For detailed descriptions of these procedures, see *Programming a BEA Tuxedo ATMI Application Using COBOL*. The COBOL interface is not described further here.

FML and VIEWS: 16-bit and 32-bit Interfaces

There are two variants of FML. The original FML interface is based on 16-bit values for the length of fields and contains information identifying fields (hence FML16). FML16 is limited to 8191 unique fields, individual field lengths of up to 64K bytes, and a total fielded buffer size of 64K. The definitions, types, and function prototypes for this interface are in `fml.h` which must be included in an application program using the FML16 interface; and functions live in `-lfml`.

A second interface, FML32, uses 32-bit values for the field lengths and identifiers. It allows for about 30 million fields, and field and buffer lengths of about 2 billion bytes. The definitions, types, and function prototypes for FML32 are in `fml32.h`; functions reside in `-lfml32`. All definitions, types, and function names for FML32 have a "32" suffix (for example, `MAXFBLEN32`, `FBFR32`, `FLDID32`, `FLDLEN32`, `F_OVHD32`, `Fchg32`, and error code `Ferror32`). Also the environment variables are suffixed with "32" (for example, `FLDTBLDIR32`, `FIELDTBLS32`, `VIEWFILES32`, and `VIEWDIR32`). For FML32, a fielded buffer pointer is of type "`FBFR32 *`", a field length has the type `FLDLEN32`, and the number of occurrences of a field has the type `FLDOCC32`. The default required alignment for FML32 buffers is 4-byte alignment.

FML16 applications that are written correctly can easily be changed to use the FML32 interface. All variables used in the calls to the FML functions must use the proper typedefs (`FLDID`, `FLDLEN`, and `FLDOCC`). Any call to `tpalloc(3c)` for an FML typed buffer should use the `FMLTYPE` definition instead of "FML". The application source code can be changed to use the 32-bit functions simply by changing the include of `fml.h` to inclusion of `fml32.h` followed by `fml1632.h`. The `fml1632.h` contains macros that convert all of the 16-bit type definitions to 32-bit type definitions, and 16-bit functions and macros to 32-bit functions and macros.

Functions are also provided to convert an FML32 fielded buffer into an FML16 fielded buffer, and vice-versa:

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

`F32to16` converts a 32-bit FML buffer to a 16-bit FML buffer. It does this by converting the buffer on a field-by-field basis and then creating the index for the fielded buffer. A field is converted by generating a FLDID from a FLDID32, and copying the field value (and field length for `string` and `carray` fields).

`dest` and *`src`* are pointers to the destination and source fielded buffers, respectively. The source buffer is not changed.

These functions can fail for lack of space; they can be re-issued after enough additional space to complete the operation has been allocated. `F16to32` converts a 16-bit FML buffer to a 32-bit FML buffer. It lives in the `fml32` library or shared object and sets `Ferror32` on error. `F32to16` lives in the `fml` library or shared object and sets `Ferror` on error. Note that both `fml.h` and `fml32.h` must be included to use these functions; `fml1632.h` may not be included in the same file.

The field types for embedded buffers (`FLD_PTR`, `FLD_FML32`, and `FLD_VIEW32`) are supported only for FML32. Buffers containing `FLD_PTR`, `FLD_FML32`, `FLD_MBSTRING`, or `FLD_VIEW32` fields cause `F32to16` to fail with an `FBADFLD` error. There is no impact when `F16to32` is called for these functions.

Note: For the remainder of this section, we describe only the 16-bit functions, without specifying the equivalent FML32 and VIEW32 functions.

Definitions of the FML Function Parameters

To simplify the specification of parameters for FML functions, a convention has been adopted for the sequence of those parameters. FML parameters appear in the following sequence.

1. For functions that require a pointer to a fielded buffer (`FBFR`), this parameter is first. If a function takes two-fielded buffer pointers (such as the transfer functions), the destination buffer comes first, followed by the source buffer. A fielded buffer pointer must point to an area that is aligned on a short boundary (or an error is returned with `Ferror` set to

FALIGNERR) and the area must be a fielded buffer (or an error is returned with Ferror set to FNOTFLD).

2. For I/O functions, a pointer to a stream follows the fielded buffer pointer.
3. For functions that need one, a field identifier (type FLDID) appears next (in the case of Fnext, it is a pointer to a field identifier).
4. For functions that need a field occurrence (type FLDOCC), this parameter comes next. (For Fnext, it is a pointer to an occurrence number.)
5. In functions in which a field value is passed to or from the function, a pointer to the beginning of the field value is given next. (It is defined as a character pointer but may be cast from any other pointer type.)
6. When a field value is passed to a function that contains a character array (carray, mbstring) field, you must specify its length as the next parameter (type FLDLEN). For functions that retrieve a field value, a pointer to the length of the retrieval buffer must be passed to the function and this length parameter is set to the length of the value retrieved.
7. A few functions require special parameters and differ from the preceding conventions. These special parameters appear after the above parameters. They are discussed in the descriptions of individual functions.
8. The following NULL values are defined for the various field types:
   - 0 for short and long
   - 0.0 for float and double
   - \0 for string (1 byte in length)
   - A zero-length string for carray or mbstring

Field Identifier Mapping Functions

Several functions allow a programmer to query field tables or field identifiers for information about fields during program execution.

Fldid

Fldid returns the field identifier for a given valid field name and loads the field name/field ID mapping tables from the field table files, if they do not already exist.

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

Here *name* is a valid field name.

The space used by the mapping tables in memory can be freed using the `Fnmid_unload, Fnmid_unload32(3fml)` function. Note that these tables are separate from the tables loaded and used by the `Fname` function.

For more information, refer to `Fldid, Fldid32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fname

`Fname` returns the field name for a given valid field identifier and loads the field ID/name mapping tables from the field table files, if they do not already exist.

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

Here *fieldid* is a valid field identifier.

The space used by the mapping tables in memory can be freed using the `Fnmid_unload, Fnmid_unload32(3fml)` function. Note that these tables are separate from the tables loaded and used by the `Fldid` function. (Refer to the *BEA Tuxedo ATMI FML Function Reference* for more information.)

For more information, refer to `Fname, Fname32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fldno

`Fldno` extracts the field number from a given field identifier.

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

Here *fieldid* is a valid field identifier.

For more information, refer to `Fldno, Fldno32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fldtype

`Fldtype` extracts the field type (an integer, as defined in `fml.h`) from a given field identifier.

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

Here *fieldid* is a valid field identifier.

The following table shows the possible values returned by Fldtype and their meanings.

Table 5-1 Field Types Returned by Fldtype

Return Value	Meaning	Field Type Name in fml.h/ fml32.h
0	Short integer	FLD_SHORT
1	Long integer	FLD_LONG
2	Character	FLD_CHAR
3	Single-precision float	FLD_FLOAT
4	Double-precision float	FLD_DOUBLE
5	Null-terminated string	FLD_STRING
6	Character array	FLD_CARRAY
9	Pointer	FLD_PTR
10	Embedded FML32 buffer	FLD_FML32
11	Embedded VIEW32 buffer	FLD_VIEW32
12	Multibyte character array	FLD_MBSTRING

For more information, refer to Fldtype, Fldtype32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Ftype

Ftype returns a pointer to a string containing the name of the type of a field given a field identifier.

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

Here *fieldid* is a valid field identifier. For example, the following code returns a pointer to one of the following strings: `short`, `long`, `char`, `float`, `double`, `string`, `carray`, `mbstring`, `FLD_PTR`, `FLD_FML32`, or `FLD_VIEW32`.

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

For more information, refer to `Ftype, Ftype32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fmkfldid

As part of an application generator, or to reconstruct a field identifier, it might be useful to make a field identifier from a type specification and an available field number. `Fmkfldid` provides this functionality.

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

Here:

- *type* is a valid type. (Specifically, it is an integer; see "Fldtype" on page 5-5 for details.)
- *num* is a field number. (It should be an unused field number to avoid confusion with existing fields.)

For more information, refer to `Fmkfldid, Fmkfldid32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Buffer Allocation and Initialization

The functions described in this section are provided for writing stand-alone FML programs. If you are using the BEA Tuxedo ATMI functions, keep in mind that for tasks such as allocating and freeing message buffers, you must call ATMI functions such as `tpalloc(3c)`, `tprealloc(3c)`, and `tpfree(3c)`, instead of FML functions such as `Falloc, Falloc32(3fml)`, `Frealloc, Frealloc32(3fml)`, and `Ffree, Ffree32(3fml)`.

Most FML functions require a pointer to a fielded buffer as an argument. The `typedef FBFR` is available for declaring such pointers, as shown in the following example:

```
FBFR *fbfr;
```

In this section, the variable *fbfr* refers to a pointer to a fielded buffer. Never attempt to declare fielded buffers themselves; declare only pointers to fielded buffers.

When a server receives a request that contains an FML buffer, it allocates space for that FML buffer and for any embedded views or buffers referenced by FLD_PTR fields. A pointer to the new FML buffer is passed to the user-written code. Once the server processing is complete, all buffers allocated when the message was received must be destroyed. The BEA Tuxedo system checks the FML buffer and all subsidiary buffers, and deletes any buffers to which it finds references. As a programmer writing server code, you should be aware of the following situations:

- If you add, change, or update a view or pointer field so that it references a buffer allocated by the server, the newly allocated buffer is deleted during the cleanup triggered when the tpreturn(3c) or tpforward(3c) function is called.
- If you change, update, or delete a field so that a buffer is no longer referenced by the FML buffer, the user-written code must free that buffer explicitly, using the tpfree(3c) function. If the buffer is not explicitly freed, the server process leaks memory.
- In some cases, the user-written code can allocate and return another buffer, rather than simply call tpreturn(3c). If this is done, the FML buffer passed to tpreturn() is freed, but any buffers referenced by FLD_PTR or FLD_VIEW32 fields are not freed.

The functions used to reserve space for fielded buffers are explained in the following text, but first we describe a function that can be used to determine whether a given buffer is, in fact, a fielded buffer.

Fielded

Fielded (or Fielded32) is used to test whether the specified buffer is fielded.

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

Fielded32 is used with 32-bit FML.

Fielded returns true (1) if the buffer is fielded. It returns false (0) if the buffer is not fielded but does not set Ferror.

For more information, refer to Fielded, Fielded32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fneeded

The amount of memory to allocate for a fielded buffer depends on the maximum number of fields the buffer will contain and the total amount of space needed for all the field values. The function `Fneeded` can be used to determine the amount of space (in bytes) needed for a fielded buffer; it takes the number of fields and the space needed for all field values (in bytes) as arguments.

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

Here:

- *F* is the number of fields.
- *V* is the space, in bytes, for field values.

The space needed for field values is computed by estimating the amount of space that is required by each field value if stored in standard structures (for example, a `long` is stored as a `long` and needs four bytes). For variable length fields, estimate the average amount of space needed for the field. The space calculated by `Fneeded` includes a fixed overhead for each field; it adds that to the space needed for the field values.

Once you obtain the estimate of space from `Fneeded`, you can allocate the desired number of bytes using `malloc`(3) and set up a pointer to the allocated memory space. For example, the following code allocates space for a fielded buffer large enough to contain 25 fields and 300 bytes of values.

```
#define NF 25
#define NV 300
extern char *malloc;
. . .
  if((fbfr = (FBFR *)malloc(Fneeded(NF, NV))) == NULL)
      F_error("pgm_name");   /* no space to allocate buffer */
```

However, this allocated memory space is not yet a fielded buffer. `Finit` must be used to initialize it.

For more information, refer to `Fneeded, Fneeded32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fvneeded

The `Fvneeded` function determines the amount of space (in bytes) needed for a `VIEW` buffer. The function takes a pointer to the name of the `VIEW` as an argument.

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

The `Fvneeded` function returns the size of the `VIEW` in number of bytes.

For more information, refer to `Fvneeded, Fvneeded32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Finit

The `Finit` function initializes an allocated memory space as a fielded buffer.

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

Here:

- *`fbfr`* is a pointer to an uninitialized fielded buffer.
- *`buflen`* is the length of the buffer, in bytes.

A call to `Finit` to initialize the memory space allocated in the previous example looks like the following code:

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

Now `fbfr` points to an initialized, empty fielded buffer. Up to `Fneeded(NF, NV)` bytes minus a small amount (`F_OVHD` as defined in `fml.h`) are available in the buffer to hold fields.

Note: The numbers used in the `malloc`(3) call (as described in the previous section) and `Finit` call must be the same.

For more information, refer to `Finit, Finit32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Falloc

Calls to `Fneeded`, `malloc`(3) and `Finit` may be replaced by a single call to `Falloc`, which allocates the desired amount of space and initializes the buffer.

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

Here:

- *F* is the number of fields.
- *V* is the space for field values, in bytes.

A call to Falloc that provides the same functionality created by the calls to Fneeded, malloc(), and Finit described in the previous three sections, must be written as follows:

```
extern FBFR *Falloc;
. . .
 if((fbfr = Falloc(NF, NV)) == NULL)
      F_error(“pgm_name”);   /* couldn't allocate buffer */
```

Storage allocated with Falloc (or Fneeded, malloc(3), and Finit) should be freed with Ffree. (See Ffree, Ffree32(3fml) in the *BEA Tuxedo ATMI FML Function Reference*.)

For more information, refer to Falloc, Falloc32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Ffree

Ffree is used to free memory space allocated as a fielded buffer. Ffree32 does not free the memory area referenced by a pointer in a FLD_PTR field.

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

Here *fbfr* is a pointer to a fielded buffer. Consider the following example:

```
#include  <fml.h>
. . .
if(Ffree(fbfr) < 0)
      F_error("pgm_name");     /* not fielded buffer */
```

Ffree is preferable to free(3), because Ffree invalidates a fielded buffer, whereas free(3) does not. It is necessary to invalidate fielded buffers because malloc(3) re-uses memory that has been freed without clearing it. Thus, if free(3) is used, malloc can return a piece of memory that looks like a valid fielded buffer, but is not.

Space for a fielded buffer may also be reserved directly. The buffer must begin on a `short` boundary. You must allocate at least `F_OVHD` bytes (defined in `fml.h`) for the buffer; if you do not, `Finit` returns an error.

The following code is analogous to the preceding example but `Fneeded` cannot be used to size the static buffer because it is not a macro:

```
/* the first line aligns the buffer */
static short buffer[500/sizeof(short)];
FBFR *fbfr=(FBFR *)buffer;
. . .
Finit(fbfr, 500);
```

Be careful *not* to enter code such as the following:

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

This code is wrong: the structure for `FBFR` is not defined in the user header files. As a result, a compilation error will be produced.

For more information, refer to `Ffree, Ffree32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fsizeof

`Fsizeof` returns the size of a fielded buffer in bytes.

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

Here *`fbfr`* is a pointer to a fielded buffer. In the following code, for example, `Fsizeof` returns the same number that `Fneeded` returned when the fielded buffer was originally allocated:

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

For more information, refer to `Fsizeof, Fsizeof32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Funused

`Funused` may be used to determine how much space is available in a fielded buffer for additional data.

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

Here *`fbfr`* is a pointer to a fielded buffer. Consider the following example:

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

Note that `Funused` does not indicate the *location*, in the buffer, of the unused bytes; only the *number* of unused bytes is specified.

For more information, refer to `Funused, Funused32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fused

`Fused` may be used to determine how much space is used in a fielded buffer for data and overhead.

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

Here *`fbfr`* is a pointer to a fielded buffer. Consider the following example:

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

Note that `Fused` does not indicate the *location*, in the buffer, of the used bytes; only the *number* of used bytes is specified.

For more information, refer to `Fused, Fused32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Frealloc

This function enables you to change the size of a buffer for which you have allocated space by calling `Falloc`.

If you have allocated space with tpalloc(3c), you must call tprealloc(3c) to reallocate that space. Being able to resize the buffer can be useful if, for example, a buffer runs out of space while a new field value is being added. Simply by calling Frealloc you can increase the size of the buffer. In other situations you may want to call Frealloc to decrease the size of the buffer.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *nf* is the new number of fields or 0.
- *nv* is the new space for field values, in bytes.

Consider the following example:

```
FBFR *newfbfr;
. . .
if((newfbfr = Frealloc(fbfr, NF+5, NV+300)) == NULL)
        F_error("pgm_name");      /* couldn't re-allocate space */
else
        fbfr = newfbfr;          /* assign new pointer to old */
```

In this case, the application needed to remember the number of fields and the number of value space bytes previously allocated. Note that the arguments to Frealloc (as with its counterpart realloc(3)) are absolute values, not increments. This example does not work if it is necessary to re-allocate space several times.

The following example shows a second way of incrementing the allocated space:

```
/* define the increment size when buffer out of space */
#define INCR    400
FBFR *newfbfr;
. . .
if((newfbfr = Frealloc(fbfr, 0, Fsizeof(fbfr)+INCR)) == NULL)
      F_error("pgm_name");       /* couldn't re-allocate space */
else
      fbfr = newfbfr;           /* assign new pointer to old */
```

You do not need to know the number of fields or the value space size with which the buffer was last initialized. Thus, the easiest way to increase the size is to use the current size plus the increment as the value space. The previous example can be executed as many times as needed

without remembering past executions or values. You do not need to call `Finit` after calling `Frealloc`.

If the amount of additional space requested in the call to `Frealloc` is contiguous to the old buffer, `newfbfr` and `fbfr` in the previous examples are the same. However, defensive programming dictates that you should declare `newfbfr` as a safeguard in case either a new value or NULL is returned. If `Frealloc` fails, do not use `fbfr` again.

Note: The buffer size can be decreased only to the number of bytes currently being used in the buffer.

For more information, refer to `Frealloc, Frealloc32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Functions for Moving Fielded Buffers

The only restriction on the location of fielded buffers is that they must be aligned on a `short` boundary. Otherwise, fielded buffers are position-independent and may be moved around freely in memory.

Fmove

If *`src`* points to a fielded buffer and *`dest`* points to an area of storage big enough to hold it, then the following code might be used to move the fielded buffer:

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

The function `memcpy`, one of the C run-time memory management functions, moves the number of bytes indicated by its third argument from the area pointed to by its second argument to the area pointed to by its first argument.

While `memcpy` may be used to copy a fielded buffer, the destination copy of the buffer looks just like the source copy. In particular, for example, the destination copy has the same number of unused bytes as the source buffer.

`Fmove` acts like `memcpy`, but does not need an explicit length (which is computed).

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

Here:

- *dest* is a pointer to the destination buffer.
- *src* is a pointer to the source fielded buffer.

In the following code, for example, Fmove checks that the source buffer is indeed a fielded buffer, but does not modify the source buffer in any way.

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

The destination buffer need not be a fielded buffer (that is, it need not have been allocated using Falloc), but it must be aligned on a short boundary (4-byte alignment for FML32). Thus, Fmove provides an alternative to Fcpy when you want to copy a fielded buffer to a non-fielded buffer. Fmove does not, however, check to make sure there is enough room in the destination buffer to receive the source buffer.

For values of type FLD_PTR, Fmove32 transfers the buffer pointer. The application programmer must manage the reallocation and freeing of buffers when the associated pointer is moved. The buffer pointed to by a FLD_PTR field must be allocated using the tpalloc(3c) call.

For more information, refer to Fmove, Fmove32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fcpy

Fcpy is used to overwrite one fielded buffer with another.

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

Here:

- *dest* is a pointer to the destination fielded buffer.
- *src* is a pointer to the source fielded buffer.

Fcpy preserves the overall buffer length of the overwritten fielded buffer and therefore is useful for expanding or reducing the size of a fielded buffer. Consider the following example:

```
FBFR *src, *dest;
. . .
```

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

Unlike Fmove, where *dest* could point to an uninitialized area, Fcpy expects *dest* to point to an initialized fielded buffer (allocated using Falloc). Fcpy also verifies that *dest* is big enough to accommodate the data from the source buffer.

Note: You cannot reduce the size of a fielded buffer below the amount of space needed for currently held data.

As with Fmove, the source buffer is not modified by Fcpy.

For values of type FLD_PTR, Fcpy32 copies the buffer pointer. The application programmer must manage the reallocation and freeing of buffers when the associated pointer is copied. The buffer pointed to by a FLD_PTR field must be allocated using the tpalloc(3c) call.

For more information, refer to Fcpy, Fcpy32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Field Access and Modification Functions

This section discusses how to update and access fielded buffers using the field types of the fields without doing any conversions. For a list of the functions that allow you to convert data from one type to another upon transfer to or from a fielded buffer, see "Conversion Functions" on page 5-43.

Fadd

The Fadd function adds a new field value to the fielded buffer.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *value* is a pointer to a new value. Its type is shown as char*, but when it is used, its type must be the same type as the value to be added (see below).
- *len* is the length of the value if its type is FLD_CARRAY or FLD_MBSTRING.

If no occurrence of the field exists in the buffer, then the field is added. If one or more occurrences of the field already exist, then the value is added as a new occurrence of the field, and is assigned an occurrence number 1 greater than the current highest occurrence. (To add a specific occurrence, Fchg must be used.)

Fadd, like all other functions that take or return a field value, expects a pointer to a field value, never the value itself.

If the field type is such that the field length is fixed (short, long, char, float, or double) or can be determined (string), the field length need not be given (it is ignored). If the field type is a character array (FLD_CARRAY or FLD_MBSTRING), the length must be specified; the length is defined as type FLDLEN. The following code, for example, gets the field identifier for the desired field and adds the field value to the buffer.

```
FLDID fieldid, Fldid;
FBFR *fbfr;
. . .
fieldid = Fldid("fieldname");
if(Fadd(fbfr, fieldid, "new value", (FLDLEN)9) < 0)
          F_error("pgm_name");
```

It is assumed (by default) that the native type of the field is a character array so that the length of the value must be passed to the function. If the value being added is not a character array, the type of value must reflect the type of the value to which it points. The following code, for example, adds a long field value.

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

For character array fields, null fields may be indicated by a length of 0. For string fields, the null string may be stored since the NULL terminating byte is actually stored as part of the field value: a string consisting of only the NULL terminating byte is considered to have a length of 1. For all other types (fixed length types), you may choose some special value that is interpreted by the application as a NULL, but the size of the value is taken from its field type (for example, a length of 4 for a long), regardless of what value is actually passed. Passing a NULL value address results in an error (FEINVAL).

For pointer fields, Fadd32 stores the pointer value. The buffer pointed to by a FLD_PTR field must be allocated using the tpalloc(3c) call. For embedded FML32 buffers, Fadd32 stores the entire FLD_FML32 field value, except for the index.

For embedded VIEW32 buffers, Fadd32 stores a pointer to a structure of type FVIEWFLD, which contains vflags (a flags field, currently unused and set to 0), vname (a character array containing the view name), and data (a pointer to the view data stored as a C structure). The application provides the vname and data to Fadd32. The FVIEWFLD structure is as follows:

```
typedef struct {
     TM32U vflags;                /* flags - currently unused */
     char vname[FVIEWNAMESIZE+1]; /* name of view */
     char *data;                  /* pointer to view structure */
} FVIEWFLD;
```

For more information, refer to Fadd, Fadd32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fappend

The Fappend function appends a new field value to the fielded buffer.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *value* is a pointer to a new value. Its type is shown as char *, but when it is used, its type must be the same type as the value to be appended (see below).
- *len* is the length of the value if its type is FLD_CARRAY or FLD_MBSTRING.

Fappend appends a new occurrence of the field *fieldid* with a value located at *value* to the fielded buffer and puts the buffer into append mode. Append mode provides optimized buffer construction for large buffers constructed of many rows of a common set of fields.

A buffer that is in append mode is restricted as to what operations may be performed on the buffer. Only calls to the following FML routines are allowed in append mode: Fappend, Findex, Funindex, Ffree, Fused, Funused and Fsizeof. Calls to Findex or Funindex end append mode.

The following example shows the construction, using Fappend, of a 500-row buffer with 5 fields per row:

```
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, like all other functions that take or return a field value, expects a pointer to a field value, never the value itself.

If the field type is such that the field length is fixed (short, long, char, float, or double) or can be determined (string), the field length need not be given (it is ignored). If the field type is a character array (FLD_CARRAY or FLD_MBSTRING), the length must be specified; the length is defined as type FLDLEN.

It is assumed (by default) that the native type of the field is a character array so that the length of the value must be passed to the function. If the value being appended is not a character array, the type of value must reflect the type of the value it points to.

For character array fields, null fields may be indicated by a length of 0. For string fields, the null string may be stored since the NULL terminating byte is actually stored as part of the field value: a string consisting of only the NULL terminating byte is considered to have a length of 1. For all other types (fixed-length types), you may choose some special value that is interpreted by the application as a NULL, but the size of the value is taken from its field type (for example, the length of 4 for a long), regardless of what value is actually passed. Passing a NULL value address results in an error (FEINVAL).

For more information, refer to Fappend, Fappend32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fchg

Fchg changes the value of a field in the buffer.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *oc* is the occurrence number of the field.
- *value* is a pointer to a new value. Its type is shown as `char *`, but when it is used, its type must be the same type as the value to be added (see "Fadd" on page 5-17).
- *len* is the length of the value if its type is `FLD_CARRAY` or `FLD_MBSTRING`.

For example, the following code changes a field of type `carray` to a new value stored in `value`:

```
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");
```

If `oc` is -1, then the field value is added as a new occurrence to the buffer. If `oc` is 0 or greater and the field is found, then the field value is modified to the new value specified. If `oc` is 0 or greater and the field is not found, then NULL occurrences are added to the buffer until the value can be added as the specified occurrence. For example, changing field occurrence 3 for a field that does not exist on a buffer causes three NULL occurrences to be added (occurrences 0, 1 and 2), followed by occurrence 3 with the specified field value. Null values consist of the NULL string "`\0`" (1 byte in length) for string and character values, `0` for long and short fields, `0.0` for float and double values, and a zero-length string for a character array.

The new or modified value is contained in `value`. If it is a character array (`FLD_CARRAY` or `FLD_MBSTRING`), its length is given in `len` (`len` is ignored for other field types). If the value pointer is NULL and the field is found, then the field is deleted. If the field occurrence to be deleted is not found, it is considered an error (`FNOTPRES`).

For pointer fields, Fchg32 stores the pointer value. The buffer pointed to by a FLD_PTR field must be allocated using the tpalloc(3c) call. For embedded FML32 buffers, Fchg32 stores the entire FLD_FML32 field value, except the index.

For embedded VIEW32 buffers, Fchg32 stores a pointer to a structure of type FVIEWFLD, which contains vflags (a flags field, currently unused and set to 0), vname (a character array containing the view name), and data (a pointer to the view data stored as a C structure). The application provides the vname and data to Fchg32. The FVIEWFLD structure is as follows:

```
typedef struct {
     TM32U vflags;               /* flags - currently unused */
     char vname[FVIEWNAMESIZE+1]; /* name of view */
     char *data;                 /* pointer to view structure */
} FVIEWFLD;
```

The buffer must have enough room to contain the modified or added field value, or an error is returned (FNOSPACE).

For more information, refer to Fchg, Fchg32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fcmp

Fcmp compares the field identifiers and field values of two fielded buffers.

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

Here *fbfr1* and *fbfr2* are pointers to fielded buffers.

The function returns a 0 if the buffers are identical; it returns a -1 on any of the following conditions:

- The fieldid of a *fbfr1* field is less than the field ID of the corresponding field of *fbfr2*.
- The value of a *fbfr1* field is less than the value of the corresponding field of *fbfr2*.
- *fbfr1* is shorter than *fbfr2*.

The following criteria are used to determine whether pointers and embedded buffers are equal:

- For pointer fields, two pointer fields are considered equal if the pointer values (addresses) are equal.

- For embedded FML32 buffers, two fields are considered equal if all field occurrences and values are equal.
- For embedded VIEW32 buffers, two fields are considered equal if the view names are the same, and if all structure member occurrences and values are equal.

`Fcmp` returns a `1` if the opposite of any of these conditions is true. For example, `Fcmp` returns `1` if the field ID of a *`fbfr2`* field is less than the field ID of the corresponding field of *`fbfr1`*.

For more information, refer to `Fcmp, Fcmp32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fdel

The `Fdel` function deletes the specified field occurrence.

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

Here:

- *`fbfr`* is a pointer to a fielded buffer.
- *`fieldid`* is a field identifier.
- *`oc`* is the occurrence number.

For example, the following code deletes the first occurrence of the field indicated by the specified field identifier:

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

If the specified field does not exist, the function returns `-1` and `Ferror` is set to `FNOTPRES`.

For pointer fields, `Fdel32` deletes the `FLD_PTR` field occurrence without changing the referenced buffer or freeing the pointer. The data buffer is treated as an opaque pointer.

For more information, refer to `Fdel, Fdel32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fdelall

`Fdelall` deletes all occurrences of the specified field from the buffer.

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

Here:

- *`fbfr`* is a pointer to a fielded buffer.
- *`fieldid`* is a field identifier.

Consider the following example:

```
if(Fdelall(fbfr, fieldid) < 0)
     F_error("pgm_name");        /* field not present */
```

If the field is not found, the function returns `-1` and `Ferror` is set to `FNOTPRES`.

For pointer fields, `Fdelall32` deletes the `FLD_PTR` field occurrence without changing the referenced buffer or freeing the pointer. The data buffer is treated as an opaque pointer.

For more information, refer to `Fdelall, Fdelall32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fdelete

`Fdelete` deletes all occurrences of all fields listed in the array of field identifiers, `fieldid[]`.

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

Here:

- *`fbfr`* is a pointer to a fielded buffer.
- *`fieldid`* is a pointer to the list of field identifiers to be deleted.

The update is done directly to the fielded buffer. The array of field identifiers does not need to be in any specific order, but the last entry in the array must be field identifier 0 (`BADFLDID`). Consider the following example:

```
#include "fldtbl.h"
FBFR *dest;
FLDID fieldid[20];
. . .
```

```
fieldid[0] = A;   /* field id for field A */
fieldid[1] = D;   /* field id for field D */
fieldid[2] = BADFLDID;   /* sentinel value */
if(Fdelete(dest, fieldid) < 0)
         F_error("pgm_name");
```

If the destination buffer has fields A, B, C, and D, this example results in a buffer that contains only occurrences of fields B and C.

`Fdelete` provides a more efficient way of deleting several fields from a buffer than using several `Fdelall` calls.

For pointer fields, `Fdelete` deletes the `FLD_PTR` field occurrence without changing the referenced buffer or freeing the pointer. The data buffer is treated as an opaque pointer.

For more information, refer to `Fdelete, Fdelete32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Ffind

`Ffind` finds the value of the specified field occurrence in the buffer.

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

Here:

- *`fbfr`* is a pointer to a fielded buffer.
- *`fieldid`* is a field identifier.
- *`oc`* is the occurrence number.
- *`len`* is the length of the value found.

In the previous declaration the return value to `Ffind` is shown as a character pointer data type (`char*` in C). The actual type of the pointer returned is the same as the type of the value to which it points.

The following code provides an example of how this function is used:

```
#include "fldtbl.h"
FBFR *fbfr;
FLDLEN len;
char* Ffind, *value;
```

```
. . .
if((value=Ffind(fbfr,ZIP,0, &len)) == NULL)
      F_error("pgm_name");
```

If the field is found, its length is returned in `len` (if `len` is NULL, the length is not returned), and its location is returned as the value of the function. If the field is not found, NULL is returned, and `Ferror` is set to `FNOTPRES`.

`Ffind` is useful for gaining "read-only" access to a field. The value returned by `Ffind` should not be used to modify the buffer. Field values should be modified only by the `Fadd` or `Fchg` function. This function does not check for occurrences of the specified field in embedded buffers.

The value returned by `Ffind` is valid only so long as the buffer remains unmodified. The value is guaranteed to be aligned on a short boundary but may not be aligned on a long or double boundary, even if the field is of that type. (See the conversion functions described later in this document for aligned values.) On processors that require proper alignment of variables, referencing the value when not aligned properly causes a system error, as shown in the following example:

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

This code should be re-written as follows:

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

For more information, refer to `Ffind, Ffind32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Ffindlast

This function finds the last occurrence of a field in a fielded buffer and returns a pointer to the field, as well as the occurrence number and length of the field occurrence.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *oc* is a pointer to the occurrence number of the last field occurrence found.
- *len* is a pointer to the length of the value found.

In the previous declaration the return value to Ffindlast is shown as a character pointer data type (char* in C). The actual type of the pointer returned is the same as the type of the value to which it points.

Ffindlast acts like Ffind, except that you do not specify a field occurrence. Instead, both the occurrence number and the value of the last field occurrence are returned. However, if you specify NULL as the value of the occurrence when calling the function, the occurrence number is not returned. This function does not check for occurrences of the specified field in embedded buffers.

The value returned by Ffindlast is valid only as long as the buffer remains unchanged.

For more information, refer to Ffindlast, Ffindlast32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Ffindocc

Ffindocc looks at occurrences of the specified field on the buffer and returns the occurrence number of the first field occurrence that matches the user-specified field value.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *value* is a pointer to a new value. Its type is shown as char*, but when it is used, its type must be the same type as the value to be added (see "Fadd" on page 5-17).
- *len* is the length of the value if its type is FLD_CARRAY or FLD_MBSTRING.

For example, the following code sets oc to the occurrence for the specified zip code:

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

Regular expressions are supported for string fields. For example, the following code sets oc to the occurrence of NAME that starts with "J":

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

Note: To enable pattern matching on strings, the fourth argument to Ffindocc must be non-zero. If it is zero, a simple string compare is performed. If the field value is not found, -1 is returned.

For upward compatibility, a circumflex (^) prefix and dollar sign ($) suffix are implicitly added to the regular expression. Thus the previous example is actually interpreted as "^(J.*)$". The regular expression must match the entire string value in the field.

For more information, refer to Ffindocc, Ffindocc32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fget

Use Fget to retrieve a field from a fielded buffer when the value is to be modified.

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

Here:

- *fbfr* is a pointer to a fielded buffer.

- *fieldid* is a field identifier.
- *oc* is the occurrence number.
- *loc* is a pointer to a buffer to copy the field value into.
- *maxlen* is a pointer to the length of the source buffer on calling the function, and a pointer to the length of the field on return.

The caller provides Fget with a pointer to a private buffer, as well as the length of the buffer. If maxlen is specified as NULL, then it is assumed that the destination buffer is large enough to accommodate the field value, and its length is not returned.

Fget returns an error if the desired field is not in the buffer (FNOTPRES), or if the destination buffer is too small (FNOSPACE). For example, the following code gets the zip code, assuming it is stored as a character array or string:

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

If the zip code is stored as a long, it can be retrieved by the following code:

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

For more information, refer to Fget, Fget32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fgetalloc

Like Fget, Fgetalloc finds and makes a copy of a buffer field, but it acquires space for the field via a call to malloc(3).

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *oc* is the occurrence number.
- *extralen* is a pointer to the additional length to be acquired on calling the function, and a pointer to the actual length acquired on return.

In the declaration above the return value to Fgetalloc is shown as a character pointer data type (char* in C). The actual type of the pointer returned is the same as the type of the value to which it points.

On success, Fgetalloc returns a valid pointer to the copy of the properly aligned buffer field; on error it returns NULL. If malloc(3) fails, Fgetalloc returns an error and Ferror is set to FMALLOC.

The last parameter to Fgetalloc specifies an extra amount of space to be acquired if, for instance, the value obtained is to be expanded before re-insertion into the fielded buffer. On success, the length of the allocated buffer is returned in extralen. Consider the following example:

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

It is the responsibility of the caller to free space acquired by Fgetalloc.

For more information, refer to Fgetalloc, Fgetalloc32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fgetlast

Fgetlast is used to retrieve the last occurrence of a field from a fielded buffer when the value is to be modified.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *oc* is a pointer to the occurrence number of the last field occurrence.
- *loc* is a pointer to a buffer to copy the field value into.
- *maxlen* is a pointer to the length of the source buffer on calling the function, and a pointer to the length of the field on return.

The caller provides Fgetlast with a pointer to a private buffer, as well as the length of the buffer. Fgetlast acts like Fget, except that you do not specify a field occurrence. Instead, both the occurrence number and the value of the last field occurrence are returned. However, if you specify NULL for occ on calling the function, the occurrence number is not returned.

For more information, refer to Fgetlast, Fgetlast32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fnext

Fnext finds the next field in the buffer after the specified field occurrence.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a pointer to a field identifier.
- *oc* is a pointer to the occurrence number.
- *value* is a pointer of the same type as the value contained in the next field.
- *len* is a pointer to the length of **value*.

A fieldid of FIRSTFLDID should be specified to get the first field in a buffer; the field identifier and occurrence number of the first field occurrence are returned in the corresponding parameters. If the field is not NULL, its value is copied into the memory location addressed by the value pointer.

The len parameter is used to determine whether value has enough space allocated to contain the field value. If the amount of space is insufficient, Ferror is set to FNOSPACE. The length of the

value is returned in the `len` parameter. If the value of the field is non-null, then the `len` parameter is also assumed to contain the length of the currently allocated space for `value`.

When the field to be retrieved is an embedded VIEW32 buffer, the *`value`* parameter points to an `FVIEWFLD` structure. The `Fnext` function populates the *`vname`* and *`data`* fields in the structure. The `FVIEWFLD` structure is as follows:

```
typedef struct {
     TM32U vflags;                /* flags - currently unused */
     char vname[FVIEWNAMESIZE+1]; /* name of view */
     char *data;                  /* pointer to view structure */
} FVIEWFLD;
```

If the field value is NULL, then the `value` and `length` parameters are not changed.

If no more fields are found, `Fnext` returns `0` (end of buffer) and `fieldid`, `occurrence`, and `value` are left unchanged.

If the `value` parameter is not NULL, the `length` parameter is also assumed to be non-NULL.

The following example reads all field occurrences in the buffer:

```
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)) {
   /* code for each field occurrence */
}
```

For more information, refer to `Fnext, Fnext32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fnum

`Fnum` returns the number of fields contained in the specified buffer, or `-1` on error.

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

Here *fbfr* is a pointer to a fielded buffer. The following code, for example, prints the number of fields in the specified buffer:

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

Each FLD_FML32 and FLD_VIEW32 field is counted as a single field, regardless of the number of fields it contains.

For more information, refer to Fnum, Fnum32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Foccur

Foccur returns the number of occurrences for the specified field in the buffer:

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.

Occurrences of a field within an embedded FML32 buffer are not counted.

Zero is returned if the field does not occur in the buffer and -1 is returned on error. For example, the following code prints the number of occurrences of the field ZIP in the specified buffer:

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

For more information, refer to Foccur, Foccur32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fpres

Fpres returns true (1) if the specified field occurrence exists. Otherwise, it returns false (0).

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *oc* is the occurrence number.

For example, the following code returns true if the field ZIP exists in the fielded buffer referenced by fbfr:

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

Fpres does not check for occurrences of the specified field within an embedded buffer.

For more information, refer to Fpres, Fpres32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fvals and Fvall

Fvals works like Ffind for string values but guarantees that a pointer to a value is returned. Fvall works like Ffind for long and short values, but returns the actual value of the field as a long, instead of as a pointer to the value.

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

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

In both functions:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is a field identifier.
- *oc* is the occurrence number.

For Fvals, if the specified field occurrence is not found, the NULL string, \0, is returned. This function is useful for passing the value of a field to another function without checking the return value. This function is valid only for fields of type string; the NULL string is automatically returned for other field types (that is, no conversion is done).

For Fvall, if the specified field occurrence is not found, then 0 is returned. This function is useful for passing the value of a field to another function without checking the return value. This function is valid only for fields of type long and short; 0 is automatically returned for other field types (that is, no conversion is done).

For more information, refer to Fvals, Fvals32(3fml) and Fvall, Fvall32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Buffer Update Functions

The functions listed in this section access and update entire fielded buffers, rather than individual fields in the buffers. These functions use, at most, three parameters:

- *dest* is a pointer to a destination fielded buffer.
- *src* is a pointer to a source fielded buffer.
- *fieldid* is a field identifier or an array of field identifiers.

Fconcat

Fconcat adds fields from the source buffer to the fields that already exist in the destination buffer.

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

Occurrences in the destination buffer are maintained (that is, they are retained and not modified) and new occurrences from the source buffer are added with greater occurrence numbers than any existing occurrences for each field. The fields are maintained in field identifier order.

Consider the following example:

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

If dest has fields A, B, and two occurrences of C, and src has fields A, C, and D, the resulting dest has two occurrences of field A (destination field A and source field A), field B, three occurrences of field C (two from dest and the third from src), and field D.

This operation fails if there is not enough space for the new fields (FNOSPACE); in this case, the destination buffer remains unchanged.

For more information, refer to Fconcat, Fconcat32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fjoin

Fjoin is used to join two fielded buffers based on matching field ID/occurrence.

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

For fields that match on field ID/occurrence, the field value is updated in the destination buffer with the value from the source buffer. Fields in the destination buffer that have no corresponding field ID/occurrence in the source buffer are deleted. Fields in the source buffer that have no corresponding field ID/occurrence in the destination buffer are not added to the destination buffer. Thus

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

Using the input buffers in the previous example results in a destination buffer that has source field value A and source field value C. This function may fail due to lack of space if the new values are larger than the old (FNOSPACE); in this case, the destination buffer will have been modified. However, if this happens, the destination buffer may be reallocated using Frealloc and the Fjoin function repeated (even if the destination buffer has been partially updated, repeating the function gives the correct results).

If joining buffers results in the removal of a pointer field (FLD_PTR), the memory area referenced by the pointer is not modified or freed.

For more information, refer to Fjoin, Fjoin32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fojoin

Fojoin is similar to Fjoin, but it does not delete fields from the destination buffer that have no corresponding field ID/occurrence in the source buffer.

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

Note that fields in the source buffer for which there are no corresponding field ID/occurrence pairs in the destination buffer are not added to the destination buffer. Consider the following example:

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

Using the input buffers from the previous example, `dest` contains the source field value A, the destination field value B, the source field value C, and the second destination field value C. As with `Fjoin`, this function can fail for lack of space (`FNOSPACE`) and can be reissued again after more space has been allocated to complete the operation.

If joining buffers results in the removal of a pointer field (`FLD_PTR`), the memory area referenced by the pointer is not modified or freed.

For more information, refer to `Fojoin, Fojoin32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fproj

`Fproj` is used to update a buffer in place so that only the desired fields are kept. (The result, in other words, is a projection on specified fields.) If updating buffers results in the removal of a pointer field (`FLD_PTR`), the memory area referenced by the pointer is not modified or freed.

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

These fields are specified in an array of field identifiers passed to the function. The update is performed directly in the fielded buffer. Consider the following example:

```
#include "fldtbl.h"
FBFR *fbfr;
FLDID fieldid[20];
. . .
fieldid[0] = A;   /* field id for field A */
fieldid[1] = D;   /* field id for field D */
fieldid[2] = BADFLDID;   /* sentinel value */
if(Fproj(fbfr, fieldid) < 0)
      F_error("pgm_name");
```

If the buffer has fields A, B, C, and D, the example results in a buffer that contains only occurrences of fields A and D. Note that the entries in the array of field identifiers do not need to be in any specific order, but the last value in the array of field identifiers must be field identifier 0 (`BADFLDID`).

For more information, refer to `Fproj, Fproj32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fprojcpy

`Fprojcpy` is similar to `Fproj` but the desired fields are placed in a destination buffer. If updating buffers results in the removal of a pointer field (`FLD_PTR`), the memory area referenced by the pointer is not modified or freed.

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

Any fields in the destination buffer are first deleted and the results of the projection on the source buffer are copied into the destination buffer. Using the above example, the following code places the results of the projection in the destination buffer:

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

The entries in the array of field identifiers may be rearranged; if the entries are not in numeric order, the field identifier array is sorted.

For more information, refer to `Fprojcpy, Fprojcpy32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fupdate

`Fupdate` updates the destination buffer with the field values in the source buffer.

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

For fields that match on field ID/occurrence, the field value is updated in the destination buffer with the value in the source buffer (like `Fjoin`). Fields on the destination buffer that have no corresponding field on the source buffer are left untouched (like `Fojoin`). Fields on the source buffer that have no corresponding field on the destination buffer are added to the destination buffer (like `Fconcat`). Consider the following example:

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

If the `src` buffer has fields A, C, and D, and the `dest` buffer has fields A, B, and two occurrences of C, the updated destination buffer contains: the source field value A, the destination field value B, the source field value C, the second destination field value C, and the source field value D.

For pointers, Fupdate32 stores the pointer value. The buffer pointed to by a FLD_PTR field must be allocated using the tpalloc(3c) call. For embedded FML32 buffers, Fupdate32 stores the entire FLD_FML32 field value, except the index.

For embedded VIEW32 buffers, Fupdate32 stores a pointer to a structure of type FVIEWFLD, which contains vflags (a flags field, currently unused and set to 0), vname (a character array containing the view name), and data (a pointer to the view data stored as a C structure). The application provides the vname and data to Fupdate32. The FVIEWFLD structure is as follows:

```
typedef struct {
     TM32U vflags;                /* flags - currently unused */
     char vname[FVIEWNAMESIZE+1]; /* name of view */
     char *data;                  /* pointer to view structure */
} FVIEWFLD;
```

For more information, refer to Fupdate, Fupdate32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

VIEWS Functions

Fvftos

This function transfers data from a fielded buffer to a C structure using a specified view description.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *cstruct* is a pointer to a structure.
- view is a pointer to a view name string.

If the named view is not found, Fvftos returns -1, and Ferror is set to FBADVIEW.

When data is being transferred from a fielded buffer to a C structure, the following rules apply:

- If a field in the fielded buffer is not mapped to a C structure member, the field is ignored.
- If a field is not in the fielded buffer, but appears in the view description and is mapped to a structure member, the corresponding null value is copied into the member.

- If a field in the fielded buffer contains data of type `string` or `carray`, characters are copied into the structure up to the size of the mapped structure member (that is, source values that are too long are truncated). If the source value is shorter than the mapped structure member, the remainder of the member value is padded with null (0) characters. String values are always terminated with a null character (even if this means truncating the value).
- If the number of occurrences of a field in the buffer is equal to the number of mapped structure members, then the fielded data is copied into the C structure.
- If the number of occurrences of a field in the buffer is greater than the number of mapped structure members, then the fielded data is ignored.
- If the number of occurrences of a field in the buffer is less than the number of mapped structure members, then the extra members are assigned the corresponding null value.

For example, the following code puts `string1` into `cust.action[0]` and `abc` into `cust.bug[0]`. All other members in the `cust` structure should contain null values.

```
#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");    /* initialize cust */
str = "string1";
Fadd(fbfr,ACTION,str,(FLDLEN)8);
str = "abc";
Fadd(fbfr,BUG_CURS,str,(FLDLEN)4);
Fvftos(fbfr,(char *)&cust,"custdb");
. . .
```

View `custdb` is defined in "VIEWS Examples" on page 6-1.

For more information, refer to `Fvftos, Fvftos32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fvstof

This function transfers data from a C structure to a fielded buffer using a specified view description.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *cstruct* is a pointer to a structure.
- *mode* is one of the following: FUPDATE, FJOIN, FOJOIN, or FCONCAT.
- *view* is a pointer to a view name string.

The transfer process obeys the rules listed under the FML function corresponding to the mode parameter: Fupdate, Fjoin, Fojoin, or Fconcat.

If the named view is not found, Fvstof returns -1, and Ferror is set to FBADVIEW.

Note: Null values are not transferred from a structure member to a fielded buffer. That is, during a structure-to-field transfer, if a structure member contains the (default or user-specified) null value defined for that member, the member is ignored.

For more information, refer to Fvftos, Fvftos32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fvnull

Fvnull is used to determine whether an occurrence in a C structure contains the null value for that field.

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

Here:

- *cstruct* is a pointer to a structure.
- *cname* is a pointer to the name of a structure member.
- *oc* is the index to a particular element.
- *view* is a pointer to a view name string.

`Fvnull` returns:

- `1` if an occurrence is null
- `0` if an occurrence is not null
- `-1` if an error occurred

For more information, refer to `Fvnull, Fvnull32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fvsinit

This function initializes all elements in a C structure to their appropriate null value.

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

Here:

- *`cstruct`* is a pointer to a structure.
- *`view`* is a pointer to a view name string.

For more information, refer to `Fvsinit, Fvsinit32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fvopt

This function allows users to change flag options at run time.

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

Here:

- *`cname`* is the name of a structure member.
- *`option`* is one of the options listed below.
- *`view`* is a pointer to a view name string.

The following list describes possible values for the *`option`* parameter.

`F_FTOS`
: Allows one-way mapping from fielded buffers to C structures. Similar to the `S` option in view descriptions.

`F_STOF`
: Allows one-way mapping from C structures to fielded buffers. Similar to the `F` option in view descriptions.

`F_BOTH`
: Allows two-way mapping between C structures and fielded buffers.

`F_OFF`
: Turns off mapping of the specified member. Similar to the `N` option in view descriptions.

Note that changes to view descriptions are not permanent. They are guaranteed only until another view description is accessed.

For more information, refer to `Fvopt, Fvopt32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fvselinit

This function initializes an individual member of a C structure to its appropriate null value. It sets the ACM of the element to 0, if the `C` flag is used in the view file; it sets the ALMs to the length of the associated null value, if the `L` flag is used in the view file.

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

Here:

- *`cstruct`* is a pointer to a structure.
- *`cname`* is a pointer to the name of a structure member.
- *`view`* is a pointer to a view name string.

For more information, refer to `Fvselinit, Fvselinit32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Conversion Functions

FML provides a set of routines that perform data conversion upon reading or writing a fielded buffer.

Generally, the functions behave like their non-conversion counterparts, except that they provide conversion from a user type to the native field type when writing to a buffer, and from the native type to a user type when reading from a buffer.

The native type of a field is the type specified for it in its field table entry and encoded in its field identifier. (The only exception to this rule is CFfindocc, which, although it is a read operation, converts from the user-specified type to the native type before calling Ffindocc.) The function names are the same as their non-conversion FML counterparts except that they include a "C" prefix.

The following field types are not supported for conversion functions: pointers (FLD_PTR), embedded FML32 buffers (FLD_FML32), and embedded VIEW32 buffers (FLD_VIEW32). If one of these field types is encountered during the execution of an FML32 conversion function, Ferror is set to FEBADOP.

CFadd

The CFadd function adds a user-supplied item to a buffer creating a new field occurrence within the buffer.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is the field identifier of the field to be added.
- *value* is a pointer to the value to be added.
- *len* is the length of the value if its type is FLD_CARRAY.
- *type* is the type of the value.

Before the field addition, the data item is converted from a user-supplied type to the type specified in the field table as the fielded buffer storage type of the field. If the source type is FLD_CARRAY (character array), the length argument should be set to the length of the array. Consider the following example:

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

If the ZIP (zip code) field were stored in a fielded buffer as a long integer, the function would convert "12345" to a long integer representation, before adding it to the fielded buffer pointed to by fbfr (note that the field value length is given as 0 since the function can determine it; the length is needed only for type FLD_CARRAY). The following code puts the same value into the fielded buffer, but does so by presenting it as a long, instead of as a string:

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

Note that the value must first be put into a variable, since C does not permit the construct &12345L. `CFadd` returns `1` on success, and `-1` on error, in which case `Ferror` is set appropriately.

For more information, refer to `CFadd, CFadd32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

CFchg

The function `CFchg` acts like `CFadd`, except that it changes the value of a field (after conversion of the supplied value).

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

Here:

- *`fbfr`* is a pointer to a fielded buffer.
- *`fieldid`* is the field identifier of the field to be changed.
- *`oc`* is the occurrence number of the field to be changed.
- *`value`* is a pointer to the value to be added.
- *`len`* is the length of the value if its type is `FLD_CARRAY`.
- *`type`* is the type of the value.

For example, the following code changes the first occurrence (occurrence 0) of field `ZIP` to the specified value, doing any needed conversion:

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

If the specified occurrence is not found, then null occurrences are added to pad the buffer with multiple occurrences until the value can be added as the specified occurrence.

For more information, refer to CFchg, CFchg32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

CFget

CFget is the conversion analog of Fget. The difference is that it copies a converted value to the user-supplied buffer.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is the field identifier of the field to be retrieved.
- *oc* is the occurrence number of the field.
- *buf* is a pointer to the post-conversion buffer.
- *len* is the length of the value if its type is FLD_CARRAY.
- *type* is the type of the value.

Using the previous example, the following code gets the value that was just stored in the buffer (regardless of which format is being used) and converts it back to a long integer:

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

If the length pointer is NULL, then the length of the value retrieved and converted is not returned.

For more information, refer to CFget, CFget32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

CFgetalloc

CFgetalloc is like Fgetalloc; you are responsible for freeing the space allocated with malloc for the returned (converted) value with free.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is the field identifier of the field to be converted.
- *oc* is the occurrence number of the field.
- *type* is the type to which the value is converted.
- *extralen* on calling the function is a pointer to the extra allocation amount; on return, it is a pointer to the size of the total allocated area.

In the declaration above, the return value to `CFgetalloc` is shown as a character pointer data type (`char*` in C). The actual type of the pointer returned is the same as the type of the value to which it points.

The previously stored value can be retrieved into space allocated automatically for you by the following code:

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

The value `extra` in the function call indicates that the function should allocate an extra 25 bytes over the amount of space sufficient for the retrieved value. The total amount of space allocated is returned in this variable.

For more information, refer to `CFgetalloc, CFgetalloc32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

CFfind

`CFfind` returns a pointer to a converted value of the desired field.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *fieldid* is the field identifier of the field to be retrieved.
- *oc* is the occurrence number of the field.
- *len* is the length of the post-conversion value.
- *type* is the type to which the value is converted.

In the previous declaration the return value to CFfind is shown as a character pointer data type (char* in C). The actual type of the pointer returned is the same as the type of the value to which it points.

Like Ffind, this pointer should be considered "readonly." For example, the following code returns a pointer to a long containing the value of the first occurrence of the ZIP field:

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

If the length pointer is NULL, then the length of the value found is not returned. Unlike Ffind, the value returned is guaranteed to be properly aligned for the corresponding user-specified type.

Note: The duration of the validity of the pointer returned by CFfind is guaranteed only until the next buffer operation, even if it is non-destructive, since the converted value is retained in a single private buffer. This differs from the value returned by Ffind, which is guaranteed until the next modification of the buffer.

For more information, refer to CFfind, CFfind32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

CFfindocc

CFfindocc looks at occurrences of the specified field on the buffer and returns the occurrence number of the first field occurrence that matches the user-specified field value after it has been converted to the type of the field identifier.

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

Here:

- *fbfr* is a pointer to a fielded buffer.

- *fieldid* is the field identifier of the field to be retrieved.
- *value* is a pointer to the unconverted matching value.
- *len* is the length of the unconverted matching value.
- *type* is the type of the unconverted matching value.

For example, the following code converts the string to the type of `fieldid` `ZIP` (possibly a `long`) and sets oc to the occurrence for the specified zip code:

```
#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");
```

If the field value is not found, `-1` is returned.

Note: Because `CFfindocc` converts the user-specified value to the native field type before examining the field values, regular expressions work only when the user-specified type and the native field type are both `FLD_STRING`. Thus, `CFfindocc` has no utility with regular expressions.

For more information, refer to `CFf indocc, CFfindocc32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Converting Strings

The following set of functions is provided to handle the case of conversion to and from a user type of `FLD_STRING`:

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

These functions call their non-string-function counterparts, providing a `type` of `FLD_STRING`, and a `len` of 0. Note that the duration of the validity of the pointer returned by `Ffinds` is the same as that described for `CFfind`.

For descriptions of these functions, see *BEA Tuxedo ATMI FML Function Reference*.

Ftypcvt

The functions `CFadd`, `CFchg`, `CFget`, `CFgetalloc`, and `CFfind` use the function `Ftypcvt` to perform the appropriate data conversion. The `Ftypcvt32` function fails for the `FLD_PTR`, `FLD_FML32`, and `FLD_VIEW32` field types. The synopsis of `Ftypcvt` usage is as follows (it does not follow the parameter order conventions).

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

Here:

- *`tolen`* is a pointer to the length of the converted value.
- *`totype`* is the type to which to convert.
- *`fromval`* is a pointer to the value from which to convert.
- *`fromtype`* is the type from which to convert.
- *`fromlen`* is the length of the from value if the from type is `FLD_CARRAY`.

`Ftypcvt` converts from the value `*fromval`, which has type `fromtype`, and length `fromlen` if `fromtype` is type `FLD_CARRAY` (otherwise `fromlen` is inferred from `fromtype`), to a value of type `totype`. `Ftypcvt` returns a pointer to the converted value, and sets `*tolen` to the converted length, upon success. Upon failure, `Ftypcvt` returns NULL. Consider the following example, in which the `CFchg` function is used:

```
CFchg(fbfr,fieldid,oc,value,len,type)
FBFR *fbfr;             /* fielded buffer */
FLDID fieldid;          /* field to be changed */
FLDOCC oc;              /* occurrence of field to be changed */
char *value;            /* location of new value */
FLDLEN len;             /* length of new value */
int type;               /* type of new value */
{
 char *convloc;         /* location of post-conversion value */
 FLDLEN convlen;        /* length of post-conversion value */
 extern char *Ftypcvt;
```

```
        /* convert value to fielded buffer type */
  if((convloc = Ftypcvt(&convlen,FLDTYPE(fieldid),value,type,len)) == NULL)
                return(-1);

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

The user may call `Ftypcvt` directly to do field value conversion without adding or modifying a fielded buffer.

For more information, refer to `Ftypcvt, Ftypcvt32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Conversion Rules

In the following list of conversion rules, `oldval` represents a pointer to the data item being converted, and `newval`, a pointer to the post-conversion value.

- When both types are identical, `*newval` is identical to `*oldval`.
- When both types are numeric, that is, if they are `long`, `short`, `float`, or `double`, the conversion is done by the C assignment operator, with proper type casting. For example, a `short` is converted to a `float` through the following code:

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

- When a numeric is being converted to a string, an appropriate `sprintf` is used. For example, a `short` is converted to a `string` through the following code:

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

- When a string is being converted to a numeric, the appropriate function (for example, `atof`, `atol`) is used, with the result assigned to a typecasted receiving location, as shown in the following example:

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

- When a type `char` is being converted to any numeric type, or when a numeric type is being converted to a `char`, the `char` is considered to be a “shorter `short`.” For example, to convert a `char` to a `float`, use the method shown in the following code:

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

  To convert a `short` to a `char`, use the method shown in the next example:

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

- A `char` is converted to a `string` by appending a NULL character. In this regard, a `char` is not a "shorter `short`." If it were, assignment would be done by converting it to a `short`, and then converting the `short` to a `string` via `sprintf`. In the same sense, a `string` is converted to a `char` by assigning the first character of the `string` to the character.
- The `carray` type is used to store an arbitrary sequence of bytes. In this sense, it can encode any user data type. Nevertheless, the following conversions are specified for `carray` types:
  - A `carray` is converted to a `string` by appending the NULL byte to the `carray`. In this sense, a `carray` can be used to store a `string`, less the overhead of the trailing NULL. (This approach does not always save space, since fields are aligned on short boundaries within a fielded buffer.) A `string` is converted to a `carray` by removing its terminating NULL byte.
  - When a `carray` is converted to any numeric, it is first converted to a `string`, and the `string` is then converted to a numeric. Likewise, a numeric is converted to a `carray`, by first being converted to a `string`, and then the `string` is converted to a `carray`.
  - A `carray` is converted to a `char` by assigning the first character of the array to the `char`. Likewise, a `char` is converted to a `carray` by assigning it as the first byte of the array, and setting the length of the array to 1.

  Note that a `carray` of length 1 and a `char` have the following differences:
  - A `char` has only the overhead of its associated `fieldid`, while a `carray` contains a length code, in addition to the associated `fieldid`.
  - A `carray` is converted to a numeric by first becoming a `string`, and then undergoing an `atoi` call; a `char` becomes a numeric by typecasting. For example, a `char` with value ASCII '1' (decimal 49) converts to a `short` of value 49; a `carray` of length 1, with the single byte an ASCII '1' converts to a `short` of value 1. Likewise a `char` 'a' (decimal 97) converts to a `short` of value 97; the `carray` 'a' converts to a `short` of value 0 (since `atoi` ("a") produces a 0 result).
- When converting to or from a `dec_t` type, the associated conversion function as described in `decimal`(3) is used (`_gp_deccvasc`, `_gp_deccvdbl`, `_gp_deccvflt`, `_gp_deccvint`, `_gp_deccvlong`, `_gp_dectoasc`, `_gp_dectodbl`, `_gp_dectoflt`, `_gp_dectoint`, and `_gp_dectolong`).

The following table summarizes the conversion rules presented in this section.

Table 5-2 Summary of Conversion Rules

src type	dest type							
-	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	-

The following table defines the entries listed in the previous table.

Table 5-3 Meanings of Entries in the Summary of Conversion Rules

Entry	Meaning
-	src and dest are the same type; no conversion required
cast	Conversion done using C assignment with type casting
sprintf	Conversion done using sprintf function
atoi	Conversion done using atoi function
atof	Conversion done using atof function
atol	Conversion done using atol function
add 0	Conversion done by concatenating NULL byte
drop 0	Conversion done by dropping terminating NULL byte
c=array[0]	Character set to first byte of array
array[0]=c	First byte of array is set to character
c=st[0]	Character set to first byte of string
st[0]=c	First byte of string set to c
d	decimal(3c) conversion function

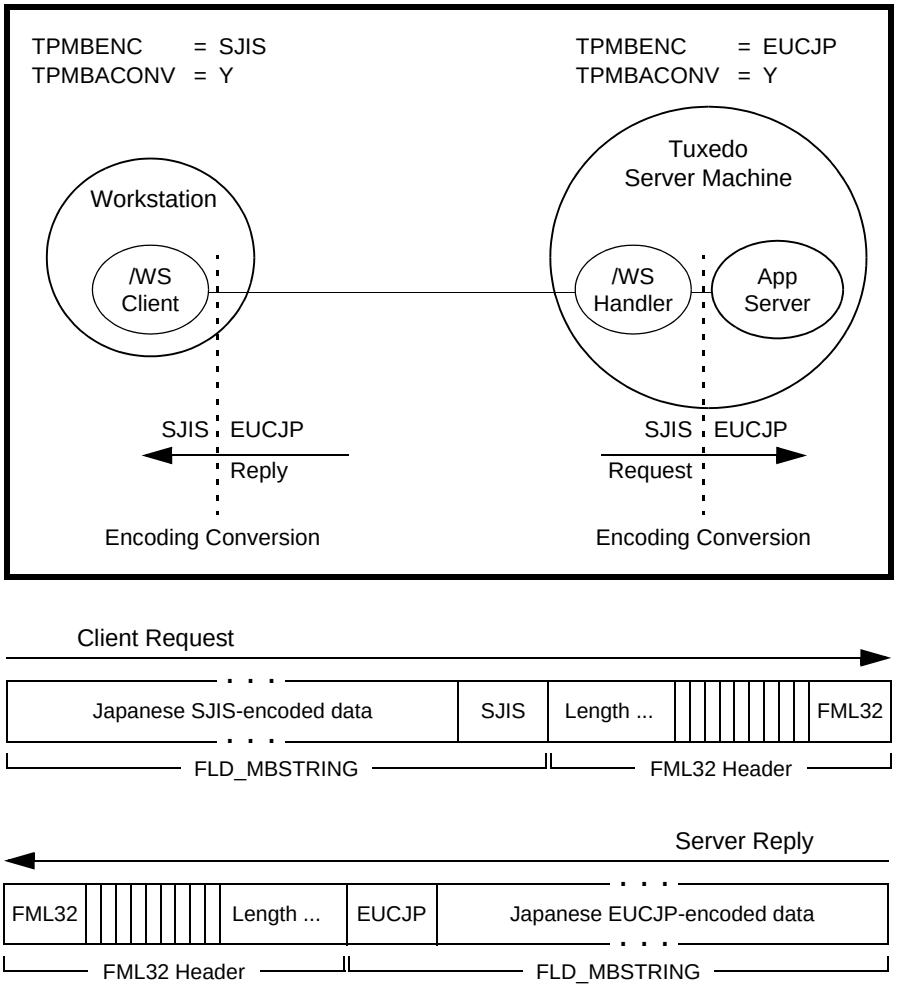
Converting FLD_MBSTRING Fields

The following set of functions is provided to handle code-set encoding conversion of data in user type of FLD_MBSTRING:

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

These functions prepare the encoding name and multibyte data information for an FLD_MBSTRING field, extract the encoding name and multibyte data information from an FLD_MBSTRING field, and convert the multibyte characters in an FLD_MBSTRING field to a named target encoding. The following figure shows through example how encoding conversion works.

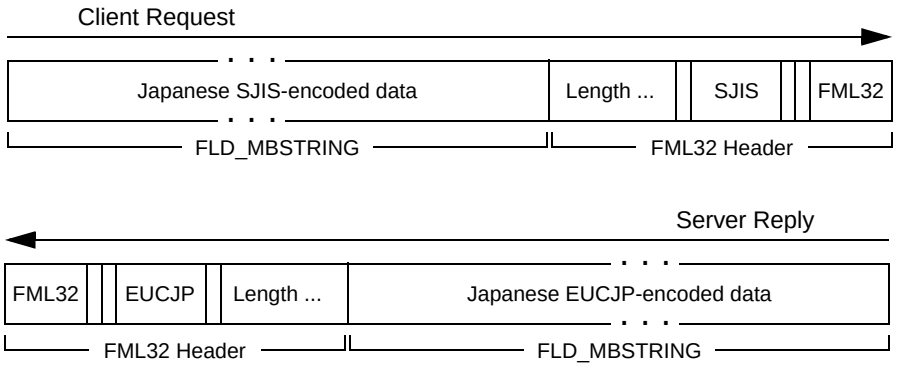
Figure 5-1 Encoding Conversion Using FML32 Buffers—Example



As indicated in the example, the `FLD_MBSTRING` field is capable of carrying information identifying the code-set *character encoding*, or simply *encoding*, of its user data. In the example, the client-request `FLD_MBSTRING` field holds Japanese user data represented by the Shift-JIS (SJIS) encoding, while the server-reply `FLD_MBSTRING` field holds Japanese user data represented by the Extended UNIX Code (EUC) encoding. The multibyte character encoding feature reads environment variables `TPMBENC` and `TPMBACONV` to determine the source encoding, the target encoding, and the state (on or off) of automatic encoding conversion.

As shown in the following figure, the FML32 typed buffer, itself, is capable of carrying information identifying the character encoding of its user data.

Figure 5-2 Using Global Encoding



For an FML32 typed buffer holding many `FLD_MBSTRING` fields, using global encoding is a more efficient way to transport multibyte user data via FML32 buffers than adding a character encoding name to each `FLD_MBSTRING` field. Using the `Fmbpack32()` function, application developers can choose global encoding or individual encoding for each `FLD_MBSTRING` field created via `Fmbpack32()`. Only one global encoding name is allowed per FML32 buffer.

The encoding conversion capability enables the underlying Tuxedo system software to convert the encoding representation of an incoming `FLD_MBSTRING` field to an encoding representation supported by the machine on which the receiving process is running. The conversion is neither a conversion between character code sets nor a translation between languages, but rather a conversion between different character encodings for the same language.

Fmbpack32

This function prepares the encoding name and multibyte data information for an `FLD_MBSTRING` field input to an FML32 typed buffer. `Fmbpack32()` is used before the `FLD_MBSTRING` field is added to an FML32 buffer via FML32 APIs.

For more information about this function, refer to the Fmbpack32(3fml) function in *BEA Tuxedo ATMI FML Function Reference*.

Fmbunpack32

This function extracts the encoding name and multibyte data information from an FLD_MBSTRING field in an FML32 typed buffer. Fmbunpack32() is used after the FLD_MBSTRING field is extracted from an FML32 buffer via FML32 APIs (Ffind32(), Fget32(), ...).

For more information about this function, refer to the Fmbunpack32(3fml) function in *BEA Tuxedo ATMI FML Function Reference*.

tpconvfmb32

This function converts the multibyte characters in an FLD_MBSTRING field in an FML32 typed buffer to a named target encoding. Specifically, tpconvfmb32() *compares* the source encoding name specified for the FLD_MBSTRING field *with* the target encoding name defined in *target_encoding*; if the encoding names are different, tpconvfmb32() converts the FLD_MBSTRING field data to the target encoding.

For more information about this function, refer to the tpconvfmb32(3fml) function in *BEA Tuxedo ATMI FML Function Reference*.

tpconvvmb32

This function converts the multibyte characters in an MBSTRING field in a VIEW32 typed buffer to a named target encoding. Specifically, tpconvvmb32() compares the source encoding name specified for the MBSTRING field with the target encoding name defined in target_encoding; if the encoding names are different, tpconvvmb32() converts the MBSTRING field data to the target encoding.

For more information about this function, refer to the tpconvvmb32(3fml) function in BEA Tuxedo ATMI FML Function Reference.

Indexing Functions

When a fielded buffer is initialized by Finit or Falloc, an index is automatically set up. This index is used to expedite fielded buffer accesses and is transparent to you. As fields are added to or deleted from the fielded buffer, the index is automatically updated.

However, when storing a fielded buffer on a long-term storage device, or when transferring it between cooperating processes, it may be desirable to save space by eliminating its index and regenerating it upon receipt. The functions described in this section may be used to perform such index manipulations.

Fidxused

This function returns the amount of space used by the index of a buffer.

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

Here *fbfr* is a pointer to a fielded buffer.

You can use this function to determine the size of the index of a buffer, and whether significant time or space can be saved by deleting the index.

For more information, refer to `Fidxused, Fidxused32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Findex

The function `Findex` may be used at any time to index an unindexed fielded buffer.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *intvl* is the indexing interval.

The second argument to `Findex` specifies the indexing interval for the buffer. If 0 is specified, the value `FSTDXINT` (defined in `fml.h`) is used. The user may ensure that all fields are indexed by specifying an interval of 1.

Note that more space may be made available in an existing buffer for user data by increasing the indexing interval, and reindexing the buffer. This represents a space/time trade-off, however, since reducing the number of index elements (by increasing the index interval), means, in general, that searches for fields will take longer. Most operations attempt to drop the entire index if they run out of space before returning a “no space” error.

For more information, refer to `Findex, Findex32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Frstrindex

This function can be used instead of `Findex` for cases in which the fielded buffer has not been altered since its index was removed.

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

Here:

- *fbfr* is a pointer to a fielded buffer.
- *numidx* is the value returned by the Funindex function.

For more information, refer to Frstrindex, Frstrindex32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Funindex

Funindex discards the index of a fielded buffer and returns the number of index entries the buffer had before the index was stripped.

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

Here *fbfr* is a pointer to a fielded buffer.

For more information, refer to Funindex, Funindex32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Example of Sending a Fielded Buffer Without an Index

To transmit a fielded buffer without its index, complete a procedure such as the following:

1. Remove the index:

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

2. Get the number of bytes to send (that is, the number of significant bytes from the beginning of the buffer):

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

3. Send the buffer without the index:

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

4. Restore the index to the buffer:

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

The index may be regenerated on the receiving side by the following statement:

```
Findex(fbfr);
```

Note that the receiving process cannot call Frstrindex because it did not remove the index itself, and the index was not sent with the file.

Note: The space used in memory by the index is not freed by calling Funindex. The Funindex function only saves space on disk or when sending a buffer to another process. Of course, you are always free to send a fielded buffer and its index to another process and avoid using these functions.

Input/Output Functions

The functions described in this section support input and output of fielded buffers to standard I/O or to file streams.

Fread and Fwrite

The I/O functions Fread and Fwrite work with the standard I/O library:

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

The stream to which—or from which—I/O is directed is determined by a FILE pointer argument. This argument must be set up using the normal standard I/O library functions.

A fielded buffer may be written into a standard I/O stream with the function Fwrite, as follows:

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

A buffer written with Fwrite may be read with Fread, as follows.

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

Although the contents of the fielded buffer pointed to by fbfr are replaced by the fielded buffer read in, the capacity of the fielded buffer (that is, the size of the buffer) remains unchanged.

Fwrite discards the buffer index, writing only as much of the fielded buffer as has been used (as returned by Fused).

Fread restores the index of a buffer by calling Findex. The buffer is indexed with the same indexing interval with which it was written by Fwrite. Fread32 ignores the FLD_PTR field type.

For more information, refer to Fread, Fread32(3fml) and Fwrite, Fwrite32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fchksum

A checksum may be calculated for verifying I/O, as follows:

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

The user is responsible for calling `Fchksum`, writing the checksum value out, along with the fielded buffer, and checking it on input. `Fwrite` does not write the checksum automatically. For pointer fields (`FLD_PTR`), the name of the pointer field in the checksum calculation (rather than the pointer or the data referenced by the pointer) is included.

For more information, refer to `Fchksum, Fchksum32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fprint and Ffprint

The `Fprint` function prints a fielded buffer on the standard output in text format.

```
Fprint(FBFR *fbfr)
```

Here `fbfr` is a pointer to a fielded buffer.

`Ffprint` is similar to `Fprint`, except that it sends text to a specified output stream, as in the following line:

```
Ffprint(FBFR *fbfr, FILE *iop)
```

Here:

- *`fbfr`* is a pointer to a fielded buffer.
- *`iop`* is a pointer of type `FILE` to the output stream.

Each of these print functions prints, for each field occurrence, the field name and the field value, separated by a tab and followed by a new line. `Fname` is used to determine the field name. If the field name cannot be determined, then the field identifier is printed. Non-printable characters in the field values for strings and character arrays are represented by a backslash followed by their two-character hexadecimal value. Backslashes occurring in the text are escaped with an extra backslash. A blank line is printed following the output of the printed buffer.

For values of type `FLD_PTR`, `Fprint32` prints the field name or field identifier and the pointer value in hexadecimal. Although this function prints pointer information, the `Fextread32` function ignores the `FLD_PTR` field type. For values of type `FLD_FML32`, `Fprint32` recursively

prints the FML32 buffer, with leading tabs added for each level of nesting. For values of type FLD_VIEW32, this function prints the VIEW32 field name and structure member name/value pairs.

For more information, refer to Fprint, Fprint32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Fextread

Fextread may be used to construct a fielded buffer from its printed format, that is, from the output of Fprint (hexadecimal values output by Fprint are interpreted properly).

```
int
Fextread(FBFR *fbfr, FILE *iop)
```

Fextread accepts an optional flag preceding the field name/field identifier specification in the output of Fprint, as shown in the following table.

Table 5-4 Fextread Flags

Flag	Indicates
+	Field should be changed in the buffer
-	Field should be deleted from the buffer
=	One field should be assigned to another
#	Comment line; ignored

If no flag is specified, the default action is to Fadd the field to the buffer.

Field values may be extended across lines by beginning each overflow line with a tab (which is later discarded). A single blank line signals the end of the buffer; successive blank lines yield a null buffer. For embedded buffers FLD_FML32 and FLD_VIEW32, Fextread generates nested FML32 buffers and VIEW32 fields, respectively. Fextread32 ignores the FLD_PTR field type.

If an error has occurred, -1 is returned, and Ferror is set accordingly. If the end of the file is reached before a blank line, Ferror is set to FSYNTAX.

For more information, refer to Fextread, Fextread32(3fml) in *BEA Tuxedo ATMI FML Function Reference*.

Boolean Expressions of Fielded Buffers

This topic includes the following sections:

This section describes the functions available for evaluating Boolean expressions in which the "variables" are the values of fields in a fielded buffer or a VIEW. Functions described in this section allow you to:

- Compile a Boolean expression into a compact form suitable for evaluation
- Evaluate a Boolean expression against a fielded buffer or a VIEW, returning a true or false answer
- Print a compiled Boolean expression

A function is provided that compiles the expression into a compact form suitable for efficient evaluation. A second function evaluates the compiled form against a fielded buffer to produce a true or false answer.

Definitions of Boolean Expressions

This section describes, in detail, the expressions accepted by the Boolean compilation function, and explains how each expression is evaluated.

The following standard C language operators are not supported:

- Shift operators: << and >>
- Bitwise "or" and "and" operators: || and &&
- Conditional operator: ?
- Prefix and postfix incrementation and decrementation operators: ++ and --
- Address and indirection operators: & and *
- Assignment operator: =

- Comma operator: ,

The following table shows the Backus-Naur Form definitions of the accepted Boolean expressions.

Table 5-5 BNF Boolean Expression Definitions

Expression	Definition
<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>
<unsigned number>	<unsigned float> | <unsigned int>
<string>	' <character> {<character>. . .} '
<field ref>	<field name> | <field name>[<field occurrence>]
<field occurrence>	<unsigned int> | <meta>
<meta>	?

The following sections describe Boolean expressions in greater detail.

Field Names and Types

The only variables allowed in Boolean expressions are field references. There are several restrictions on field names. Names are made up of letters and digits; the first character must be a letter. The underscore (_) counts as a letter; it is useful for improving the readability of long variable names. Up to 30 characters are significant. There are no reserved words.

For a fielded buffer evaluation, any field that is referenced in a Boolean expression must exist in a field table. This implies that the `FLDTBLDIR` and `FIELDTBLS` environment variables are set, as described in "Setting Up Your Environment for FML and VIEWS" on page 3-1 before using the Boolean compilation function. The field types used in Booleans are those allowed for FML fields: `short`, `long`, `float`, `double`, `char`, `string`, and `carray`. Along with the field name, the field type is kept in the field table. Thus, the field type can always be determined.

For a VIEW evaluation, any field that is referenced in a Boolean expression must exist as a C structure element name, not the associated fielded buffer name, in the VIEW. This implies that the `VIEWDIR` and `VIEWFILES` environment variables are set, as described in "Setting Up Your Environment for FML and VIEWS" on page 3-1 before using the Boolean compilation function. The field types used in Booleans are those allowed for FML VIEWS: `short`, `long`, `float`, `double`, `char`, `string`, `carray`, plus `int` and `dec_t`. Along with the field name, the field type is kept in the view definition. Thus, the field type can always be determined.

Strings

A string is a group of characters within single quotes. The ASCII code for a character may be substituted for the character via an escape sequence. An escape sequence takes the form of a backslash followed by exactly two hexadecimal digits. **This convention differs from the C language convention of using a hexadecimal escape sequence that starts with `\x`.**

As an example, consider `'hello'` and `'hell\\6f'`. They are equivalent strings because the hexadecimal code for an 'o' is `6f`.

Octal escape sequences and escape sequences such as `\n` are not supported.

Constants

Numeric integer and floating point constants are accepted, as in C. (Octal and hexadecimal constants are not recognized.) Integer constants are treated as `long`s and floating point constants are treated as `double`s. (Decimal constants for the `dec_t` type are not supported.)

How a Boolean Expression Is Converted for Evaluation

To evaluate a Boolean expression, the Boolean compiler performs the following conversions:

- It converts `short` and `int` values to `long`s.
- It converts `float` and `decimal` values to `double`s.
- It converts characters to `string`s.
- To compare a non-quoted string within a field to a numeric, it converts the string to a numeric value.
- To compare a `constant` (that is, a quoted) string to a numeric, it converts the numeric to a string, and does a lexical comparison.
- To compare a `long` and a `double`, it converts the `long` to a `double`.

Description of Boolean Primary Expressions

Boolean expressions are built from primary expressions, which can be any of the following:

- `field name`—a field name
- `field name[constant]`—a field name and a constant subscript
- `field name[?]`—a field name and the '`?`' subscript
- `constant`—a constant
- `(expression)`—an expression in parentheses

A field name or a field name followed by a subscript is a primary expression. The subscript indicates which occurrence of the field is being referenced. The subscript may be either an integer constant, or `?` indicating any occurrence; the subscript cannot be an expression. If the field name is not subscripted, field occurrence 0 is assumed.

If a field name reference appears without an arithmetic, unary, equality, or relational operator, then its value is the long integer value 1 if the field exists and 0 if the field does not exist. This may be used to test the existence of a field in the fielded buffer regardless of field type. (Note that there is no `*` indirection operator.)

A constant is a primary expression. Its type may be `long`, `double`, or `carray`, as discussed in the conversion section.

A parenthesized expression is a primary expression for which the type and value are identical to those of the unadorned expression. Parentheses may be used to change the precedence of operators, which is discussed in the next section.

Description of Boolean Expression Operators

The following table lists the Boolean expression operators in descending order of precedence.

Table 5-6 Boolean Expression Operators

Type	Operators
Unary	+, -, !, ~
Multiplicative	*, /, %
Additive	+, -
Relational	< , >, <=, >=, ==, !=
Equality and matching	==, !=, %%, !%
Exclusive OR	^
Logical AND	&&
Logical OR	||

The operators classified as the same operator type have equal precedence. The following sections discuss each operator type in detail. As in C, you can override the precedence of operators by using parentheses.

Unary Operators Used in Boolean Expressions

The following unary operators are recognized:

- Unary plus operator: +
- Unary minus operator: -
- The one's complement operator: ~
- Logical not operator: !

Expressions in which unary operators are used group right-to-left:

```
+ expression
- expression
~ expression
! expression
```

The unary plus operator has no effect on the operand; it is recognized and ignored. The result of the unary minus operator is the negative of its operand. The usual arithmetic conversions are performed. Unsigned entities do not exist in FML and thus cause no problems with this operator.

The result of the logical negation operator is 1 if the value of its operand is 0, and 0 if the value of its operand is non-zero. The type of the result is `long`.

The result of the one's complement operator is the one's complement of its operand. The type of the result is `long`.

Multiplicative Operators Used in Boolean Expressions

The multiplicative operators—`*`, `/`, and `%`—group left-to-right. The usual arithmetic conversions are performed:

```
expression * expression
expression / expression
expression % expression
```

The binary `*` operator indicates multiplication. The `*` operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary `/` operator indicates division. When positive integers are divided, truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative.

The binary `%` operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be `float` or `double`.

Additive Operators Used in Boolean Expressions

The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions are performed:

```
expression + expression
expression - expression
```

The result of the + operator is the sum of the operands. The + operator is associative and expressions with several additions at the same level may be rearranged by the compiler. The operands must not both be `string`s; if one is a `string`, it is converted to the arithmetic type of the other.

The result of the - operator is the difference of the operands. The usual arithmetic conversions are performed. The operands must not both be `string`s; if one is a `string`, it is converted to the arithmetic type of the other.

Equality and Match Operators Used in Boolean Expressions

These operators group left-to-right:

```
expression == expression
expression != expression
expression %% expression
expression !% expression
```

The == (equal to) and the `!=` (not equal to) operators yield 0 if the specified relation is false and 1 if it is true. The type of the result is `long`. The usual arithmetic conversions are performed.

The %% operator takes, as its second expression, a regular expression against which it matches its first expression. The second expression (the regular expression) must be a quoted string. The first expression may be an FML field name or a quoted string. This operator yields a 1 if the first expression is fully matched by the second expression (the regular expression). The operator yields a 0 in all other cases.

The `!%` operator is the *not regular expression match* operator. It takes exactly the same operands as the %% operator, but yields exactly the opposite results. The relationship between %% and `!%` is analogous to the relationship between == and `!=`.

The regular expressions allowed are described on the `tpsubscribe(3c)` reference page in the *BEA Tuxedo ATMI C Function Reference*.

Relational Operators Used in Boolean Expressions

These operators group left-to-right:

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `long`. The usual arithmetic conversions are performed.

Exclusive OR Operator Used in Boolean Expressions

The `^` operator groups left-to-right:

`expression ^ expression`

It returns the bitwise exclusive OR function of the operands. The result is always a `long`.

Logical AND Operator Used in Boolean Expressions

`expression && expression`

The `&&` operator groups left-to-right. It returns 1 if both its operands are non-zero; otherwise, it returns 0. The `&&` operator guarantees left-to-right evaluation. However, it is *not* guaranteed that the second operand is not evaluated if the first operand is 0; this is different from the C language. The operands need not have the same type. The result is always a `long`.

Logical OR Operator Used in Boolean Expressions

The `||` operator groups left-to-right:

`expression || expression`

It returns `1` if either of its operands is non-zero; otherwise, it returns `0`. The `||` operator guarantees left-to-right evaluation. However, it is not guaranteed that the second operand is not evaluated if the first operand is non-zero; this is different from the C language. The operands need not have the same type, and the result is always a `long`.

Sample Boolean Expressions

The following field table defines the fields used for the sample Boolean expressions:

```
EMPID    200    carray
SEX      201    char
AGE      202    short
DEPT     203    long
SALARY   204    float
NAME     205    string
```

Boolean expressions always evaluate to either true or false. The following example is true if both of the following conditions are true:

- Field occurrence 2 of EMPID exists and begins with the characters “123.”
- The age field (occurrence 0) appears and is less than 32.

```
"EMPID[2] %% '123.*' && AGE < 32"
```

This example uses a constant integer as a subscript to EMPID. In the following example, the ? subscript is used, instead:

```
"PETS[?] == 'dog'"
```

This expression is true if PETS exists and any occurrence of it contains the characters “dog”.

Boolean Functions

The following sections describe the various functions that take Boolean expressions as arguments.

Fboolco and Fvboolco

Fboolco compiles a Boolean expression for FML and returns a pointer to an evaluation tree:

```
char *
Fboolco(char *expression)
```

Here *expression is a pointer to an expression to be compiled. This function fails if any of the following field types is used: FLD_PTR, FLD_FML32, or FLD_VIEW32. If one of these field types is encountered, Ferror is set to FEBADOP.

Fvboolco compiles a Boolean expression for a VIEW and returns a pointer to an evaluation tree:

```
char *
Fvboolco(char *expression, char *viewname)
```

Here *expression is a pointer to an expression to be compiled, and *viewname is a pointer to the view name for which the fields are evaluated.

Space is allocated using malloc(3) to hold the evaluation tree. For example, the following code compiles a Boolean expression that checks whether the FIRSTNAME field is in the buffer, whether it begins with ‘J’ and ends with ‘n’ (such as “John” or “Joan”), and whether the SEX field is equal to ‘M’.

```
#include "<stdio.h>"
#include "fml.h"
extern char *Fboolco;
char *tree;
. . .
if((tree=Fboolco("FIRSTNAME %% 'J.*n' && SEX == 'M'")) == NULL)
  F_error("pgm_name");
```

The first and second characters of the tree array form the least significant byte and the most significant byte, respectively, of an unsigned 16-bit quantity that gives the length, in bytes, of the entire array. This value is useful for copying or otherwise manipulating the array.

Because the evaluation tree produced by `Fboolco` is used by the Boolean functions described in the following sections, it is not necessary to recompile the expression constantly.

Use the `free`(3) function to free the space allocated to an evaluation tree when the Boolean expression will no longer be used. Compiling many Boolean expressions without freeing the evaluation tree when it is no longer needed may cause a program to run out of data space.

For more information, refer to `Fboolco, Fboolco32, Fvboolco, Fvboolco32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fboolpr and Fvboolpr

`Fboolpr` prints a compiled expression to the specified file stream. The expression is fully parenthesized, as it was parsed (as indicated by the evaluation tree).

```
void
Fboolpr(char *tree, FILE *iop)
```

Here:

- `*tree` is a pointer to a Boolean tree previously compiled by `Fboolco`.
- `*iop` is a pointer of type `FILE` to an output file stream.

`Fvboolpr` prints a compiled expression to the specified file stream.

```
void
Fvboolpr(char *tree, FILE *iop, char *viewname)
```

Here:

- `*tree` is a pointer to a Boolean tree previously compiled by `Fvboolco`.

- `*iop` is a pointer of type `FILE` to an output file stream.
- `*viewname` is the name of the view whose fields are used.

This function is useful for debugging.

Executing `Fboolpr` on the expression compiled above produces the following results:

```
(((FIRSTNAME[0]) %% ('J.*n')) && ((SEX[0]) == ('M')))
```

For more information, refer to `Fboolpr, Fboolpr32, Fvboolpr, Fvboolpr32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

Fboolev and Ffloatev, Fvboolev and Fvfloatev

These functions evaluate a fielded buffer against a Boolean expression.

```
int Fboolev(FBFR *fbfr,char *tree)
double Ffloatev(FBFR *fbfr,char *tree)
```

Here:

- *`fbfr`* is the fielded buffer referenced by an evaluation tree produced by `Fboolco`.
- *`tree`* is a pointer to an evaluation tree that references the fielded buffer pointed to by `fbfr`.

The VIEW equivalents are as follows:

```
int
Fvboolev(FBFR *fbfr,char *tree,char *viewname)

double
Fvfloatev(FBFR *fbfr,char *tree,char *viewname)
```

`Fboolev` returns true (1) if the fielded buffer matches the Boolean conditions specified in the evaluation tree. This function does not change either the fielded buffer or the evaluation tree. Using the evaluation tree compiled above, the following code prints “Buffer selected”:

```
#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` and `Ffloatev32` are similar to `Fboolev`, but return the value of the expression as a `double`. For example, the following code prints “6.6”:

```
#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));
  }
}
```

If `Fboolev` is used instead of `Ffloatev` in the previous example, a 1 is printed.

For more information, refer to `Fboolev, Fboolev32, Fvboolev, Fvboolev32(3fml)` and `Ffloatev, Ffloatev32, Fvfloatev, Fvfloatev32(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

VIEW Conversion to and from Target Format

A VIEW can be converted to and from a target record format. The default target format is that of IBM System/370 COBOL records.

Fvstot, Fvftos and Fcodeset

The following functions convert targets:

```
long
Fvstot(char *cstruct, char *trecord, long treclen, char *viewname)
```

```
long
Fvttos(char *cstruct, char *trecord, char *viewname)
```

```
int
Fcodeset(char *translation_table)
```

The `Fvstot` function transfers data from a C structure to a target record type. The `Fvttos` function transfers data from a target record to a C structure. `trecord` is a pointer to the target record. *`cstruct`* is a pointer to a C structure. `viewname` is a pointer to the name of a compiled view description. The `VIEWDIR` and `VIEWFILES` environment variables are used to find the directory and file containing the compiled view description.

To convert an FML buffer to a target record, complete the following procedure.

1. Call `Fvftos` to convert the FML buffer to a C structure.
2. Call `Fvstot` to convert to a target record.

To convert a target record to an FML buffer, complete the following procedure.

1. Call `Fvttos` to convert to a C structure.
2. Call `Fvstof` to convert the structure to an FML buffer.

The default target is that of IBM/370 COBOL records. The default data conversion is done as shown in the following table.

Table 5-7 Data Conversion from a Structure to a Record

Struct	Record
`float`	`COMP-1`
`double`	`COMP-2`
`long`	`S9(9) COMP`
`short`	`S9(4) COMP`
`int`	`S9(9) COMP` or `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`	Character array

No filler bytes are provided between fields in an IBM/370 record. The COBOL SYNC clause should not be specified for any data items that are a part of the structure corresponding to the view. An integer field is converted to either a four-byte or two-byte integer, depending on the size of integers on the machine on which the conversion is done. A string field in the view must be terminated with a null when converting to and from the IBM/370 format. The data in a `carray` field is passed unchanged; no data translation is performed.

Packed decimals exist in the IBM/370 environment as two decimal digits packed into one byte with the low-order half byte used to store the sign. The length of a packed decimal may be 1 to 16 bytes with storage available for 1 to 31 digits and a sign. Packed decimals are supported in C structures using the `dec_t` field type. The `dec_t` field has a defined size consisting of two numbers separated by a comma. The number to the left of the comma is the total number of bytes occupied by the decimal. The number to the right is the number of digits to the right of the decimal point. The following formula is used for conversion:

`dec_t(`*m, n*`) <=> S9(2*`*m*`-(`*n*`+1))V9(`*n*`)COMP-3`

Decimal values may be converted to and from other data types (such as `int`, `long`, `string`, `double`, and `float`) using the functions described in `decimal(3c)`.

See the `Fvstof, Fvstof32(3fml)` for a description of the default character conversion of ASCII to EBCDIC, and vice-versa.

An alternate character translation table can be used at run time by calling `Fcodeset`. The `translation_table` must point to 512 bytes of binary data. The first 256 bytes of data are interpreted as the ASCII-to-EBCDIC translation table. The second 256 bytes of data are interpreted as the EBCDIC-to-ASCII table. Any data after the 512th byte is ignored. If the pointer is NULL, the default translation is used.

For more information, refer to `Fvstot, Fvttos(3fml)` in *BEA Tuxedo ATMI FML Function Reference*.

CHAPTER 6

FML and VIEWS Examples

This topic includes the following sections:

- VIEWS Examples
- FML Examples in bankapp

VIEWS Examples

The VIEWS examples provided in this section are unrelated to the example FML program that appears later in this section.

Sample Viewfile

The following listing is a sample of a viewfile containing a source view description, `custdb`.

Listing 6-1 Sample Viewfile

```
# BEGINNING OF VIEWFILE
VIEW custdb
#  /* This is a comment */
#  /* This is another comment */
#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
```

Sample Field Table

The following listing is a sample of a field table needed to compile the view in the last section.

Listing 6-2 Sample Field Table

```
# 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  -     -
```

Sample Header File Produced by viewc

The following listing shows a header file produced by the view compiler. Assume that the viewfile in the earlier section was used as input to `viewc`.

Listing 6-3 Sample Header File Produced by viewc

```
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" */
};
```

Sample Header File Produced by mkfldhdr

The following listing shows a header file produced from a field table file by `mkfldhdr`. Assume that a field table file containing the definitions of the fields shown in the previous examples was used as input to `mkfldhdr`.

Listing 6-4 Sample Header File Produced by mkfldhdr(1)

```
/* custdb.flds.h as generated by mkfldhdr from a field table:      */
/*       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   */
```

Sample COBOL COPY File

The following listing shows the COBOL COPY file, `CUSTDB.cbl`, produced by `viewc` with the `-C` option.

Listing 6-5 Sample COBOL COPY File

```
*       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).
```

For a sample COBOL program that includes a COBOL COPY file produced by `viewc -C`, see *Programming a BEA Tuxedo ATMI Application Using COBOL*.

Sample VIEWS Program

The following program is an example of the use of VIEWS to map a structure to a fielded buffer. The environment variables discussed in "Setting Up Your Environment for FML and VIEWS" on page 3-1 must be properly set for this program to work.

Information on compiling FML programs can be found on the `compilation(5)` reference page in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

Listing 6-6 Sample VIEWS Program

```
/* sample VIEWS program */
#include stdio.h>
#include "fml.h"
#include "custdb.flds.h"     /* field header file shown in */
/* “Sample Header File Produced by viewc” listing */
#include "custdb.h"          /* C structure header file produced by */
/* viewc shown in “Sample Field Table” listing */
#define NF 800
#define NV 400
extern Ferror;
main()
{
  /* declare needed program variables and FML functions */
    FBFR *fbfr,*Falloc();
    void F_error();
```

```
   char *str, *cstruct, buff[100];
   struct custdb cust;

 /* allocate a fielded buffer */
   if ((fbfr = Falloc(NF,NV)) == NULL) {
           F_error("sample.program");
           exit(1);
 }

    /* initialize str pointer to point to buff     */
    /* copy string values into buff, and           */
    /* Fadd values into some of the fields in fbfr */

    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");

/* Print out the current contents of the fbfr */

    printf("fielded buffer before:\n"); Fprint(fbfr);

/* Put values in the C structure */

   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;

 /* Update the fbfr with the values in the C structure */
 /* using the custdb view description. */

 if (Fvstof(fbfr,cstruct,FUPDATE,"custdb") < 0) {
           F_error("custdb");
           Ffree(fbfr);
           exit(1);
 }

 /* Note that the following would transfer */
 /* data from fbfr to cstruct */
 /*
 if (Fvftos(fbfr,cstruct,"custdb") < 0) {
           F_error("custdb");
           Ffree(fbfr);
           exit(1);
 } */

 /* print out the values in the C structure and */
 /* the values in the fbfr       */

 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);
}
```

Example of VIEWS in bankapp

`bankapp` is a sample application distributed with the BEA Tuxedo system. It includes two files in which a VIEWS structure is used. The structure in the example is one that does not map to an FML buffer, so FML functions are not used to get data into or out of the structure members.

`$TUXDIR/apps/bankapp/audit.c` is a client program that uses command-line options to determine how to set up a service request in a `VIEW` typed buffer.

The code in the server `$TUXDIR/apps/bankapp/BAL.ec` accepts the service request and shows the fields from a `VIEW` buffer being used to formulate ESQL statements.

See Also

- `viewc, viewc32(1)` in *BEA Tuxedo Command Reference*
- `mkfldhdr, mkfldhdr32(1)` in *BEA Tuxedo Command Reference*

FML Examples in bankapp

`bankapp` is a sample application distributed with the BEA Tuxedo system. The servers

`ACCT.ec`
`BTADD.ec`
`TLR.ec`

show FML functions being used to manipulate data in FML typed buffers that have been passed to the servers from `bankclt`, the bankapp client.

Note that in these servers the ATMI functions `tpalloc(3c)` and `tprealloc(3c)`—rather than the FML functions `Falloc, Falloc32(3fml)` and `Frealloc, Frealloc32(3fml)`—are used to allocate message buffers.

APPENDIX A

FML Error Messages

The following table lists the error codes, numbers, and messages that you might see if an error occurs during the execution of an FML program.

Table A-1 FML Error Codes and Messages

Error Code	#	Error Message
FALIGN	1	Fielded buffer not aligned
FNOTFLD	2	Buffer not fielded
FNOSPACE	3	No space in fielded buffer
FNOTPRES	4	Field not present
FBADFLD	5	Unknown field number or type
FTYPERR	6	Illegal field type
FEUNIX	7	UNIX system call error
FBADNAME	8	Unknown field name
FMALLOC	9	`malloc` failed
FSYNTAX	10	Bad syntax in Boolean expression
FFTOPEN	11	Cannot find or open field table

Table A-1 FML Error Codes and Messages (Continued)

Error Code	#	Error Message
FFTSYNTAX	12	Syntax error in field table
FEINVAL	13	Invalid argument to function
FBADTBL	14	Destructive concurrent access to field table
FBADVIEW	15	Cannot find or get view
FVFSYNTAX	16	Syntax error in viewfile
FVFOPEN	17	Cannot find or open viewfile
FBADACM	18	ACM contains negative value
FNOCNAME	19	`cname` not found
FEBADOP	20	Invalid field type