THE ENTERPRISE MIDDLEWARE SOLUTION

BEA TUXEDO

FML Programmer's Guide

BEA TUXEDO Release 6.5
Document Edition 6.5
February 1999

BEA TUXEDO FML Programmer's Guide

Document Edition	Date	Software Version
6.5	February 1999	BEA TUXEDO Release 6.5

Contents

CHAPTER

1 Introduction

About This Guide and FML

This chapter of the BEA TUXEDO FML Programmer’s Guide is intended to give you an idea of what the guide is about, how the Field Manipulation Language fits into the BEA TUXEDO system, and how you might get the most out of this document.

This guide assumes that you are familiar with the BEA TUXEDO system, and that you have at least read the *BEA TUXEDO Application Development Guide*.

What Is FML?

FML is a set of C language functions for defining and manipulating storage structures called `fielded buffers`, that 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 the reverse as well). VIEWS lets you perform lengthy manipulations of data in structures rather than in fielded buffers; applications will run faster if data is transferred to structures for manipulation. VIEWS allows the data independence of fielded buffers to be combined with the efficiency and simplicity of classic record structures.

The original FML and VIEW interfaces allowed for 16-bit field identifiers, field lengths, field occurrences, and record lengths. A newer FML32 and VIEW32 interface allows for larger identifiers (32-bit), field lengths, field occurrences, and record lengths. The interfaces are similar but the type definitions, header files, function names, and command names are suffixed with “32”.

How Does FML Fit into the BEA TUXEDO System?

Within the BEA TUXEDO system, FML functions are used to manipulate fielded buffers.

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 programs that receive input in fielded buffers from data entry programs, you will need to use FML functions.

Even if you elect to develop your own applications programs for handling user input and output (and you do not use DES, the BEA TUXEDO-supplied data entry system), or if programs are written to pass messages between processes, FML may still be the way you choose to deal with fielded buffers passed between these programs.

Who Is This Document For?

This is a guide for programmers who need to learn how to use FML functions. As a programmer using FML, you might be working on BEA TUXEDO system data entry programs, or other programs requiring inter-process communication of fielded data. This guide also provides information for users of applications that make use of FML with regard to setting up the environment correctly.

This guide gives detailed information about the features of FML and how the different FML functions are used.

Prerequisites

To make full use of this guide, you should be familiar with the following:

- The UNIX System environment—We assume, for example, that you do not need a definition of a shell command or an environment variable, and that you understand what is meant by a UNIX System file or running a process in the background.
- The C programming language—The functions and macros that make up FML are intended to be incorporated in C language programs, so we assume you have previously spent some time developing C programs. If you are using VIEWS in COBOL (that is, COBOL records), little, if any, C language knowledge is needed.
- The BEA TUXEDO system—We assume, even if you have not yet worked on a BEA TUXEDO application, that you at least have an understanding of what the BEA TUXEDO system is intended to do, and that you have read about the application development environment in the *BEA TUXEDO Programmer's Guide* or the *BEA TUXEDO COBOL Guide*.

What Does This Document Include?

- Concepts and Definitions—Several definitions are extracted from the *BEA TUXEDO Glossary*. These explain ideas and terms that are used in the guide. In some cases these things may sound familiar to you, but are used in a possibly unfamiliar way. You will find these definitions toward the end of this chapter.
- An Overview of FML—Chapter 2 offers an overview of the software. If you have not used FML functions before, you may find it helpful to read through the overview to get a general idea of how things work.
- Setup and Customization—Chapter 3 gives you the information you need to set up the environment variables, directory structure, and files that are required by the BEA TUXEDO system in general, and FML in particular. This chapter also shows you how to customize your installed FML software.
- Defining and Using FML Fielded Buffers and VIEWS—Chapter 4 outlines the use of the FML and VIEWS software, and how to set up your C or COBOL language programs to use the software.

- FML Field Manipulation Functions—Chapter 5 deals with how to use the FML and VIEWS functions to manipulate data.
- Code Fragments—There are illustrations throughout Chapters 4 and 5 that show you examples of the functions as they might be used in a C program. (COBOL examples are given in the *BEA TUXEDO COBOL Guide*.) Chapter 6 has a VIEWS example and refers to other examples that are part of `bankapp`, the sample application distributed with the BEA TUXEDO system.

What Other FML Documentation Is There?

In addition to this guide, documentation on FML function calls can be found in Section 3fml of the *BEA TUXEDO Reference Manual*. Three other pages in Section 5 of the *BEA TUXEDO Reference Manual* are relevant to FML: `compilation`(5), `field_tables`(5), and `viewfile`(5).

Table 1-1 Section 5 reference pages

Reference Page	Description
`compilation(5)`	describes how to compile application programs
`field_tables(5)`	describes the structure of FML field tables
`viewfile(5)`	describes the structure of VIEW description files

Concepts and Definitions

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 the 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

Fields in FML and fielded buffers are typed. They can be any of the standard C language types: `short`, `long`, `float`, `double`, and `char`. Two other types are also supported: `string` (a series of characters ending with a null character) and `carray` (character arrays). The corresponding types in COBOL are `COMP-5`, `COMP-1`, `COMP-2` and `PIC X`. A C packed decimal type is also supported in VIEWS for integration with `COBOL COMP-3`.

VIEWS

VIEWS is a part 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 a 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.

BEA TUXEDO System 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. An additional difference of BEA TUXEDO VIEW buffers is that they can be totally unrelated to an FML fielded buffer. In this text the emphasis is very much on the idea that a VIEW is a structured version of an FML record. In other texts, such as the *BEA TUXEDO Programmer's Guide*, you will find the emphasis shifts to thinking of VIEWs as another of the available BEA TUXEDO buffer types.

2 Overview

Introduction

This chapter begins by describing two ways in which the idea of fielded records or fielded buffers can be handled: through structured records and through FML records. It goes on to tell you about the features of the Field Manipulation Language, and to describe the circumstances under which you might want to make use of them.

A comparison of FML records with traditional structured records clearly shows the advantages of using fielded buffers throughout an application.

Dividing Records into Fields

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 the BEA TUXEDO system there are two ways to divide records into fields:

- through C language data structures or COBOL records
- through fielded buffers

Structures

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 the field is shown by the 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 sex, it could be done with a structure like the following:

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

where 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 `sex` 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->sex` 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 SEX                              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 `SEX IN MYREC` can be used to access the fields.

Possible Disadvantages of Structures

While this way of representing data is widely used and 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; this most often results in wasted space, since not all fields will always contain a value and since fields tend to be sized to hold the largest likely entry.

Fielded Buffers

Fielded buffers provide another way of 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, and all access to fields is through function calls.

Fielded buffers can be used throughout the BEA TUXEDO system as the standard method of representing data sent between cooperating processes.

Implementing Fielded Buffers with FML

Fielded buffers are created, updated, accessed, input, and output via the Field Manipulation Language (FML). FML has two main objectives:

- To provide a convenient and standard discipline for creating and manipulating fielded buffers
- To provide data independence to programs making use of fielded buffers

FML is implemented as a library of functions and macros that can be called from C programs. There are three major groups of FML functions:

- A set of functions for creating, updating, accessing, and manipulating fielded buffers
- A set of functions for converting data from one type to another upon input to (or output from) a fielded buffer structure
- A set of functions for 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.

To use VIEWS, your program must know the format of incoming fielded buffer data. This is done through 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

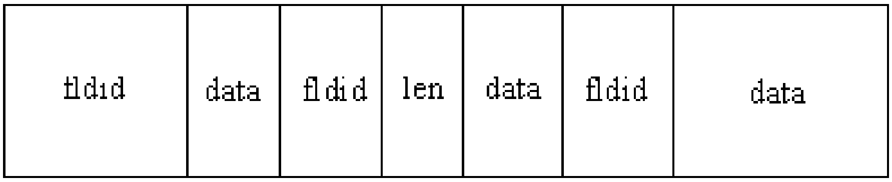
This section describes the features of FML and VIEWS, and gives you some preliminary information about how you can use them in application programs.

Fielded Buffer Structure

A fielded buffer, as mentioned earlier, 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, `fldid` is followed by a length indicator. The buffer can be represented as a sequence of `fldid`/data pairs, with `fldid`/length/data triples for variable-length items. Figure 2-1 illustrates this.

Figure 2-1 A fielded buffer



In the header file that is `#include`'d whenever FML functions are used (`fml.h` or `fml32.h`), field identifiers are `typedef`'d 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`, `character`, `string`, and `carray` (character array). These types are `#define`'d in `fml.h` (or `fml32.h`) as shown in Listing 2-1.

Listing 2-1 FML field types as defined 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 */
```

`FLD_STRING` and `FLD_CARRAY` are both 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` 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` 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.

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. See “VIEWS Features” later in this chapter

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 the `decimal`(3c) reference page. Within the COBOL program, the field can be accessed directly using a packed decimal (`COMP-3`) definition. Since 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. (See “Field Names and Identifiers” in Chapter 4 for a detailed description of field identifiers.) This allows you to reference fields in a program without using the field name, which may change.

There are two ways in which identifiers are assigned (mapped) to field names:

- Through field table files (which are ordinary UNIX files)
- Through 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. (See “Loading the Field Tables” in Chapter 4.)

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 Chapter 4.

Compile-Time: Header Files

`mkfldhdr`(1) (or `mkfldhdr32`) is provided to make header files out of field table files. These header files are `#include`'d in C programs, and provide another way to map field names to field identifiers: at compile-time.

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 what field names your program needs, `#include`-ing your field table header file(s) saves some data space and means your program can get to the task at hand more quickly.

However, since this method resolves mappings at compile-time, it should not be used if the field name/identifier mappings in the application are likely to change. This topic is covered in more detail in Chapter 4, “Field Definition and Use.”

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.

A function, `Funindex`, is provided to do that. When the fielded buffer is read from disk (or received from a sending process), the index can be explicitly reconstructed with the function `Findex`.

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

Multiply Occurring Fields

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 make 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

Often, application programs receive a fielded buffer from another source (from a user's terminal, from a database record, and so on) and the values of one or more fields will determine the next action taken by the application program. FML provides several functions that create boolean expressions on fielded buffers or VIEWs and determine if 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 if 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

VIEWS 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 gain in processing efficiency by using the VIEWS functions to transfer fielded buffer data from the buffer to a C structure before you manipulate it. This is because the FML functions for manipulating fields in a buffer require more processing time than C functions. Then, when you finish processing the data in the C structure, you can transfer it back to the fielded buffer and send it on to another program.

The VIEWS software 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.
- A view compiler, viewc(1) (or `viewc32`), 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.

- 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`, and `FCONCAT`; these modes are similar to the ones supported by the following FML functions: `Fupdate()`, `Fjoin()`, `Fojoin()`, and `Fconcat()`.
- At runtime, 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 FML supported types can be used in view descriptions, and 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.
- Multiple field occurrences are supported.
- User-specified and default null values in view descriptions are supported.
- As described earlier for fielded buffers, functions exist to compile and evaluate boolean expressions against application data in a VIEW.

A source viewfile is an ordinary UNIX text file that contains one or more source view descriptions. Source viewfiles are used as input to the view compiler, `viewc`(1) (or `viewc32`), which compiles the source view descriptions and stores them in object viewfiles.

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 procedures provided are `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 Chapter 3, “Setup,”). The first object viewfile with the specified name will be loaded. Subsequent object viewfiles with the same name, if any, are ignored.

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

Multiply Occurring Fields in VIEWS

Since VIEWS is concerned with moving fields between fielded buffers and C structures or COBOL records, it has to deal with the possibility of multiply occurring 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 receiving array has more elements than there are occurrences in the fielded buffer, the extra elements are assigned the (default or user-specified) null value. If there are more occurrences in the buffer than there are 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

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; 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.

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

The error codes that can be produced by an FML function are described on each FML reference page in Section 3fml of the *BEA TUXEDO Reference Manual*.

CHAPTER

3 Setup

Introduction

This chapter deals with the setup of the FML environment. Before you can begin to work with FML fielded buffers, or use the VIEWS functions that move fields between structures and fielded buffers, you have to take care of such details as setting environment variables appropriate for your application. These activities are described in this chapter.

Directory Structure

The delivered FML software will reside in a subtree of the local file system. Several of the FML modules assume that the structure of this subtree is as described in this section. It is assumed that the environment variable `TUXDIR` is set to the full pathname of the installation directory for the BEA TUXEDO system software. The sub-directories are:

- `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, `$TUXDIR/lib/libfml.`*`suffix`* and `$TUXDIR/lib/libgp.`*`suffix`* should be included 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 95, Windows NT, and OS/2; it is part of the BEA TUXEDO system DLL for platforms that use dynamic link libraries.)

C application software using FML must include the following header files in this order:

```
#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

Several environment variables are used by FML and VIEWS. This section gives a summary of their use.

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 (described in Chapter 4, “Field Definition and Use,”):

- `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 file names. 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`.

In addition to the ones needed by FML (`FLDTBLDIR` and `FIELDTBLS`), two environment variables are used by VIEWS functions:

- `VIEWFILES`—This variable should contain a comma separated list of object viewfiles for the application. Files given as full pathnames are used as is; files listed as relative path names are searched for through the list of directories specified by the `VIEWDIR` variable (see below). `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 file names. Its usage is similar to the `PATH` environment variable. If `VIEWDIR` is not set or is null, then its value is taken to be the current directory. `VIEWDIR32` is used for `VIEW32`.

CHAPTER

4 Field Definition and Use

Introduction

Before you can begin to work with FML fielded buffers, or use the VIEWS functions that move fields between structures and fielded buffers, certain details must be taken care of, such as:

- defining fields
- making field definitions available to applications programs (through field table files and mapping functions at run-time, or C header files at compile time)
- compiling source view descriptions into object view descriptions, and generating corresponding C header files and COBOL COPY files

These and related activities are described in this chapter.

Defining Fields

This section discusses

- how fields are defined in field tables for run-time use
- the available functions for run-time use with the field table files

Field Names and Identifiers

A field identifier (`fieldid`) is defined (`typedef`'d) 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).

Field numbers are restricted to be between 1 and 8191, inclusive, for FML, and between 1 and 33,554,431, inclusive, for FML32. Field number 0 and the corresponding field identifier 0 is 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 numbering convention adopted by the BEA TUXEDO system is as follows:

- field numbers 1-100 are reserved for system use
- field numbers 101-8191 are for application-defined fields with FML, and field numbers 101-33,554,431 for FML32.

The mappings between field identifiers and field names are contained in either field table files or field header files. Using field table files requires that you convert field name references in C programs with the mapping functions described later in this chapter; field header files allow the C preprocessor (`cpp`(1) in UNIX reference manuals) to resolve name-to-fieldid 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, which are to be used (`FLDTBLDIR32` and `FIELDTBLS32` are used for FML32). These should be set as described in Chapter 3, "Setup."

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.

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 $ ignored by the mapping functions but are passed through (without the $) to header files generated by mkfldhdr(1); for example, this would allow the application to pass C comments, what strings, etc. to the generated C header file; they 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 following 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, and is specified as one of the following: char, string, short, long, float, double, carray.
  - 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 fields in each group will be numbered 501 and 701, respectively.

Listing 4-1 A UNIX 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 Functions

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

`Fldid` maps its argument, a field name, to a `fieldid`:

```
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:

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

The identifier-to-name mapping is rarely used; that is, it is rare that one has a field identifier and wants to know the corresponding name. One place where the field identifier-to-field name mapping could be used is in a buffer print routine where you want to display, in an intelligible form, the contents of a fielded buffer.

Loading the 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` (`Ferror32` is set for FML32).

To recover the data space used by the field tables loaded by `Fldid()`, the user may unload all of the files by a call to the `Fnmid_unload()` function.

The function `Fname()` acts in a fashion similar to `Fldid`(), but 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 function `Fidnm_unload()`.

Both mapping functions and other FML functions that use run-time mapping require `FIELDTBLS` and `FLDTBLDIR` to be set properly. Otherwise, default values are used. (See Chapter 3, “Setup,” for the defaults.)

Field Header Files

The command `mkfldhdr`(1) (or `mkfldhdr32`) converts field tables, as described above, into header files 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.

The header file need not be used if the run-time mapping functions are used as described in the next sub-section. 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 will not be propagated to the service routine (that is, it will use the mappings it has already compiled).

`mkfldhdr`(1) translates each field-table specified in the `FIELDTBLS` environment variable to a corresponding header file, whose name is formed by concatenating a `.h` suffix to the field-table name. The resulting files are created, by default, in the current directory. The user may specify a creation directory to `mkfldhdr`(1) by specifying a `-d` option followed by the name of the directory in which you want the header files to reside. For example,

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

or

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

will produce the include files `maskftbl.h`, `DBftbl.h` and `miscftbl.h` in the current directory by processing `${FLDTBLDIR}/maskftbl`, `${FLDTBLDIR}/DBftbl` and `${FLDTBLDIR}/miscftbl`. The command

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

will process the sample input field-table files and produce the same output files, but will place them in the directory given by `${FLDTBLDIR}`.

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 does not apply to the run-time mapping functions). In this case, `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,

```
mkfldhdr myfields
```

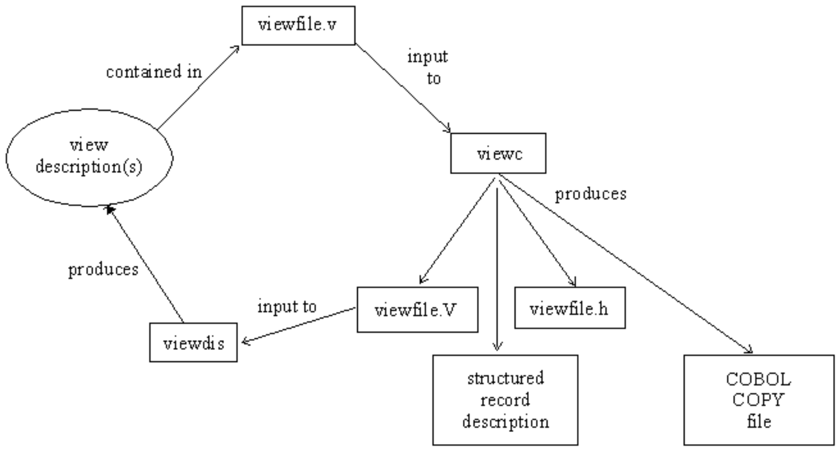
will convert the field table file `myfields` to a field header file `myfields.h`, and place it in the current directory.

Mapping Fields to C Structures and COBOL Records

As mentioned in Chapter 2, "Overview," FML VIEWS is a mechanism that allows the exchange of data between fielded buffers and C structures or COBOL records. This capability is provided since it is usually more efficient to perform lengthy manipulations on C structures with C functions than on fielded buffers with FML functions. It also provides a way for a COBOL program to send and receive messages with a C program that handles FML fielded records.

This section discusses VIEWS and how to use it to provide fielded buffer/structure mappings. The figure below shows the various components of VIEWS and how they relate to one another. Each component is explained in the following sections.

Figure 4-1 Views



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 in turn be used as input to the view disassembler (`viewdis` or `viewdis32`), which translates the object view descriptions back into their source format (for verification or editing).

You create and edit the source view descriptions (or edit the output of `viewdis`) only; compiled view descriptions are not readable by an editor.

Besides the actual view description(s), viewfiles can 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 lets you pass C comments, `what` strings, etc., to C header files produced by the view compiler; they are not passed through to the COBOL copy files.

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; the name can have a maximum of 33 alphanumeric characters (including the underscore character); when used with `tpalloc`(3c), the maximum number of characters is 16.
- 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.

Thus, a source view description has the general structure shown in Listing 4-2.

Listing 4-2 Source View Description

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

In Listing 4-2:

- *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`, `dec_t`; if type is '`-`', the type of the member is defaulted to the type of `fbname`.
- *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; 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); 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 list of options, separated by commas, or '-' meaning no options are set; see below for a discussion of *flag* options.
- *size* is the size of the member if the type is `string`, `carray`, or `dec_t`; otherwise '`-`' should be specified, and the view compiler will compute the size. For `string` or `carray`, it 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, `size` is two numbers separated by a comma, the first being the number of bytes in the decimal value (it must be greater than 0 and less than 10) and the second being the number of decimal places to the right of the decimal point (it must be greater than 0 and less than two times the number of bytes minus one).
- `null` is the user-specified null value or '`-`' to indicate the default null value for that field; see below for a discussion of null values.

flag Options

The following is a list of the options that can be specified as the `flag` element of a member description in a view description:

`C`

This option specifies that an additional structure member, called the associated count member (ACM), be generated, in addition to the structure member described in the member description. When transferring data 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, then the first N non-null fields are transferred to the fielded buffer. If N is greater than the dimension of the array, it then 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` would be declared as follows:

```
short C_parts;
```

For the COBOL COPY file, the type is declared to be `PIC S9(4) USAGE COMP-5` for FML and `PIC S9(9) USAGE COMP-5` for FML32, and its name is generated as `C-`*`cname`*.

Note: It is possible for the generated ACM name to conflict with structure members whose names begin with a `C_` prefix. Such conflicts will be reported by the view compiler, and are considered fatal errors by the compiler. For example, if a structure member has the name `C_parts`, it would conflict 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 `string` or `carray` 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 was truncated on transfer. When transferring data 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 since `carray` fields 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 to be 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 whose names begin with an `L_` prefix. Such conflicts will be reported by the view compiler, and are considered fatal errors by the compiler. For example, if a structure member has the name `L_parts`, it will conflict 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 can be used to allocate fillers in C structures or COBOL records. This option 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). Note that a member whose value is null will not be 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.

Null Values

Null values are used in VIEWS to indicate empty C structure or COBOL record members. Default null values are provided, and 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 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.

View Compiler

`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 runtime to effect the actual mapping of data. At runtime, 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 (see Figure 4-1).

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: Users of the BEA TUXEDO system Workstation feature in an MS-DOS environment will notice that the object viewfile is given a .VV suffix.

viewc C Header Files

Header files created by the view compiler (viewc) can be used 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" */
};
```

COBOL COPY Files

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 will look 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 file name is automatically converted to upper case by the view compiler. The COPY file would be included in a COBOL program as follows.

```
01 MYREC COPY TEST.
```

The output in the resulting COPY files is more fully described in the *BEA TUXEDO COBOL Guide*.

View Disassembler

The view disassembler 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. It is usually used to verify the correctness of an object view description.

The command line looks like the following.

```
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(s), and 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 original source view description, but this does not affect the correctness of the object file.

CHAPTER

5 Field Manipulation Functions

Introduction

This chapter describes all of the FML and FML VIEWS functions, with the exception of the run-time mapping functions described in Chapter 4, “Field Definition and Use.” In this chapter you will learn:

- ♦ FML parameter conventions
- ♦ how to use various field identifier mapping functions
- ♦ how to allocate and initialize fielded buffers
- ♦ how to move fielded buffers
- ♦ how to access and modify fielded buffers
- ♦ how to update fielded buffers
- ♦ how to map fielded buffers to C structures
- ♦ how to perform type conversions on data transferred to or from fielded buffers
- ♦ how to use indexing functions
- ♦ how to use input/output functions
- ♦ how to construct boolean expressions to make program decisions based on the contents of fielded buffers

For COBOL programs, the FML functions are not directly available. A procedure called `FINIT` is available to initialize a record for receiving FML data, and `FVSTOF` and `FVFTOS` are available to convert from a COBOL record to an FML buffer and back. These are described in detail in the *BEA TUXEDO COBOL Guide*. The COBOL interface will not be described further in this chapter.

FML/FML32 and VIEW/VIEW32

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`; and functions live 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`. Also note that the default required alignment for FML32 buffers is 4-byte alignment.

Existing 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` 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 to 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 allocating enough additional space to complete the operation. `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.

For the remainder of this chapter, rather than continuing to give both the FML and FML32 names, and VIEW and VIEW32 names, the 16-bit functions will be described.

FML Parameters

To make it easier to remember the parameters for the FML functions, a convention has been adopted for the sequence of function 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 the input/output 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 where a field value is passed to or from the function, a pointer to the beginning of the field value is given next (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) 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 and will be discussed in the individual function descriptions.
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); and a zero-length string for carray.

Field Identifier Mapping Functions

Several functions allow the 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/fieldid mapping tables from the field table files, if they do not already exist:

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

where *`name`* is a valid field name.

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

Fname

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

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

where *`fieldid`* is a valid field identifier.

The space used by the mapping tables in memory can be freed using the `Fidnm_unload` or `Fidnm_unload32` function. Note that these tables are separate from the tables loaded and used by the `Fldid` function.

Fldno

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

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

where *`fieldid`* is a valid field identifier.

Fldtype

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

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

where *`fieldid`* is a valid field identifier.

Table 5-1 shows the possible values returned by `Fldtype` and their meanings.

Table 5-1 Field Types Returned by Fldtype

Return Value	Meaning
0	short integer
1	long integer
2	character
3	single-precision float
4	double-precision float
5	null-terminated string
6	character array

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)
```

where *`fieldid`* is a valid field identifier.

For example:

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

returns a pointer to one of the following strings: `short`, `long`, `char`, `float`, `double`, `string`, or `carray`.

Fmkfldid

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

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

where

- *`type`* is a valid type (an integer; see `Fldtype`, above)
- *`num`* is a field number (it should be an unused field number, to avoid confusion with existing fields)

Buffer Allocation and Initialization

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

```
FBFR *fbfr;
```

In this chapter, the variable *`fbfr`* will be used to mean a pointer to a fielded buffer.

Never attempt to declare fielded buffers themselves, only pointers to them. The functions used to reserve space for fielded buffers are explained in the following pages, but first we will 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 and does not set `Ferror` in this case.

Fneeded

The amount of memory to allocate for a fielded buffer depends on the maximum number of fields it 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)
```

where

- ♦ `F` is the number of fields
- ♦ `V` is the space for field values, in bytes

The space needed for field values is computed by estimating the amount of space that would be required by each field value if stored in standard structures (e.g., a long is stored as a long and needs four bytes, etc.). For variable length fields, you should 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 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.

Finit

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

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

where

- ♦ *`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 above would look like the following:

```
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) (from the previous section) and `Finit` calls must be the same.

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)
```

where

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

A call to `Falloc` that would replace the examples above would look like the following:

```
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`.

Remember that when using the BEA TUXEDO system, the ATMI functions `tpalloc`, `tprealloc`, and `tpfree` must be used to allocate and free message buffers, rather than the FML functions `Falloc`, `Frealloc`, and `Free`.

Ffree

`Ffree` is used to free memory space allocated as a fielded buffer.

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

where

- ♦ *`fbfr`* is a pointer to a fielded buffer

For example:

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

`Ffree` is recommended as opposed to `free`(3), because `Ffree` will invalidate a fielded buffer whereas `free`(3) will 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) were used, it would be possible for `malloc` to 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. The user must allocate at least `F_OVHD` bytes (defined in `fml.h`) for the buffer or an error will be returned from `Finit`.

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

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

It should be emphasized that the following code is quite wrong:

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

The structure for FBFR is not defined in the user header files so this will result in a compilation error.

Fsizeof

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

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

where

- ♦ *`fbfr`* is a pointer to a fielded buffer

For example:

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

`Fsizeof` returns the same number that `Fneeded` returned when the fielded buffer was originally allocated.

Funused

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

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

where

- ♦ *`fbfr`* is a pointer to a fielded buffer

For example:

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

Note that `Funused` does not indicate where in the buffer the unused bytes are, only the number of unused bytes.

Fused

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

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

where

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

For example:

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

Note that `Fused` does not indicate where in the buffer the used bytes are, only the number of used bytes.

Frealloc

At some point, the buffer may run out of space, such as during the addition of a new field value. `Frealloc` can be used to increase (or decrease) the size of the buffer:

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

where

- *`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

For 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 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 will not work if space needs to be re-allocated 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 */
```

Note that 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 above example could be executed as many times as needed without remembering past executions or values. The user need not 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 examples above will be the same. However, defensive programming dictates that the user should declare `newfbfr` as a safeguard against the case where either a new value or NULL is returned. If `Frealloc` fails, do not use `fbfr` again.

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

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 freely around 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 might be used to move the fielded buffer:

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

The function `memcpy`, part of the C runtime 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 (it is computed):

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

where

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

For example:

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

`Fmove` checks that the source buffer is indeed a fielded buffer, but does not modify the source buffer in any way.

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` (see below) when it is desired to copy a fielded buffer to a non-fielded buffer, but `Fmove` does not check to make sure there is enough room in the destination buffer to receive the source buffer.

Fcpy

`Fcpy` is used to overwrite one fielded buffer with another:

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

where

- *`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; thus, `Fcpy` is useful for expanding or reducing the size of a fielded buffer. For 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`) and also checks to see that it 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`.

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. The functions that allow you to convert data from one type to another upon transfer to/from a fielded buffer are listed under "Conversion Functions" later in this chapter.

Fadd

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

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

where

- ♦ *`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`

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, the length must be specified; the length is defined as type `FLDLEN`. For example:

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

gets the field identifier for the desired field and adds the field value to the buffer.

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 it points to; for instance, the following 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 will be taken from its field type (e.g., length of four for a *long*) regardless of what value is actually passed. Passing a NULL value address will result in an error (`FEINVAL`).

Fappend

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

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

where

- ♦ *`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`

`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` will end append mode. The following example shows the construction of a 500 row buffer with 5 fields per row using `Fappend`.

```
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, 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 will be taken from its field type (e.g., length of four for a *long*) regardless of what value is actually passed. Passing a NULL value address will result in an error (`FEINVAL`).

Fchg

`Fchg` changes the value of a field in the buffer.

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

where

- ♦ *`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`)
- ♦ *`len`* is the length of the value if its type is `FLD_CARRAY`

For example, to change 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 will cause 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, 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`).

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

Fcmp

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

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

where

- ♦ *`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`*

`Fcmp` returns a 1 if any of the reverse set of the above conditions is true (for example, if the field id of a *`fbfr2`* field is less than the field id of the corresponding field of *`fbfr1`*, etc.).

Fdel

The `Fdel` function deletes the specified field occurrence.

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

where

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

For example,

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

deletes the first occurrence of the field indicated by the specified field identifier. If it does not exist, the function returns `-1` (`Ferror` is set to `FNOTPRES`).

Fdelall

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

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

where

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

For example:

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

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

Fdelete

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

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

where

- ♦ *`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`). For 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 will result in a buffer that contains only occurrences of fields B and C.

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

Ffind

Ffind finds the value of the specified field occurrence in the buffer:

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

where

- ♦ *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 declaration above 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 it points to.

An example of the use of the function is:

```
#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 value modification should only be done by the functions Fadd or Fchg.

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 will cause a system error, as 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;
```

and should be re-written as:

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

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)
```

where

- ♦ `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 declaration above 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 it points to.

`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 for occurrence on calling the function, the occurrence number will not be returned.

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

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;)
```

where

- ♦ *`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`)
- ♦ *`len`* is the length of the value if type `carray`

For example,

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

would set `oc` to the occurrence for the specified zip code.

Regular expressions are supported for string fields. For example,

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

would set `oc` to the occurrence of NAME that starts with “J”.

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

For upward compatibility, a circumflex (^) and dollar sign ($) are implied to surround the regular expression; thus, the above example is actually interpreted as “^(J.*)$”. This means that the regular expression must match the entire string value in the field.

Fget

`Fget` should be used 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)
```

where

- ♦ *`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,

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

gets the zip code assuming it is stored as a character array or string. If it is stored as a long, then it would be retrieved by:

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

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)
```

where

- *`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 it points to.

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 (`Ferror` is set to `FMALLOC`).

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

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)
```

where

- *`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 will not be returned.

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)
```

where

- ♦ *`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 if `value` has enough space allocated to contain the field value (`Ferror` is set to `FNOSPACE` if it does not); and, the length of the value is returned in the `len` parameter. Note that 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`.

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 */
}
```

Fnum

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

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

where

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

For example:

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

would print the number of fields in the specified buffer.

Foccur

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

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

where

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

Zero is returned if the field does not occur in the buffer and `-1` is returned on error. For example:

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

would print the number of occurrences of the field `ZIP` in the specified buffer.

Fpres

`Fpres` returns true (1) if the specified field occurrence exists and false (0) otherwise:

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

where

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

For example:

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

would return true if the field `ZIP` exists in the fielded buffer pointed to by `fbfr`.

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 a pointer to the value.

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

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

where 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 (i.e., 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 (i.e., no conversion is done).

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`*, *`src`*, and *`fieldid`*, where

- *`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 (i.e., 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).

In 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 resultant `dest` will have 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 will fail if there is not enough space to contain the new fields (`FNOSPACE`); in this case, the destination buffer remains unchanged.

Fjoin

`Fjoin` is used to join two fielded buffers based on matching fieldid/occurrence.

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

For fields that match on fieldid/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 fieldid/occurrence in the source buffer are deleted. Fields in the source buffer that have no corresponding fieldid/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 will result 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 re-allocated using `Frealloc` and the `Fjoin` function repeated (even if the destination buffer has been partially updated, repeating the function will give the correct results).

Fojoin

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

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

Note that fields that exist in the source buffer that have no corresponding fieldid/occurrence in the destination buffer are not added to the destination buffer. For example:

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

Using the input buffers from the previous example, `dest` will contain 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 re-issued again after allocating more space to complete the operation.

Fproj

`Fproj` is used to update a buffer in place so that only the desired fields are kept (in other words, the result is a projection on specified fields).

```
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. For 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`).

Fprojcpy

`Fprojcpy` is similar to `Fproj` but the projection is done into a destination buffer.

```
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,

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

will place the results of the projection in the destination buffer. The entries in the array of field identifiers may be re-arranged; the field identifier array is sorted if they are not in numeric order.

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 fieldid/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`). For 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 will contain: 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.

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)
```

where

- ♦ *`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 transferring data 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 will be copied into the structure up to the size of the mapped structure member (i.e., source values that are too long will be truncated). If the source value is shorter than the mapped structure member, the remainder of the member value will be padded with null (0) characters. String values will always be 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,

```
#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");
. . .
```

would put “string1” into `cust.action[0]` and “abc” into `cust.bug[0]`. All other members in the `cust` structure should contain null values.

View `custdb` is defined in “VIEWS Examples” in Chapter 6.

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)
```

where

- ♦ *`fbfr`* is a pointer to a fielded buffer
- ♦ *`cstruct`* is a pointer to a structure
- ♦ *`mode`* is one of the following: `FUPDATE`, `FJOIN`, `FOJOIN`, `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`, and `Fconcat`, described in this chapter).

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.

Fvnull

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

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

where

- ♦ *`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
```

Fvsinit

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

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

where

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

Fvopt

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

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

where

- ♦ *`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

Possible values for the *`option`* parameter are:

`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.

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)
```

where

- ♦ *`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

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 they have a "C" prefix.

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)
```

where

- *`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 of type `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. For 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 fragment:

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

puts the same value into the fielded buffer, but does so by presenting it as a long, instead of as a string. 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.

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)
```

where

- *`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 of type `carray`
- *`type`* is the type of the value

For example,

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

would change the first occurrence (occurrence 0) of field `ZIP` to the specified value, doing any needed conversion.

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.

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)
```

where

- *`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 of type `carray`
- *`type`* is the type of the value

Using the previous example,

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

would get the value that was just stored in the buffer, no matter what format, and convert it back to a long integer. If the length pointer is NULL, then the length of the value retrieved and converted is not returned.

CFgetalloc

`CFgetalloc` is like `Fgetalloc`; you are responsible for freeing the `malloc`'d space for the returned (converted) value with `free`:

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

where

- ♦ *`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 it points to.

The previously stored value could 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 not only allocate enough space for the retrieved value but an additional 25 bytes and the total amount of space allocated will be returned in this variable.

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)
```

where

- *`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 declaration above 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 it points to.

Like `Ffind`, this pointer should be considered read only. For example:

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

would return a pointer to a long containing the value of the first occurrence of the ZIP field. 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.

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 (it is converted to the type of the field identifier).

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

where

- ♦ *`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,

```
#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");
```

would convert the string to the type of `fieldid` `ZIP` (possibly a long) and set `oc` to the occurrence for the specified zip code. If the field value is not found, `-1` is returned.

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

Converting Strings

A set of functions (`Fadds`(3fml), `Fchgs`(3fml), `Fgets`(3fml), `Fgetsa`(3fml) and `Ffinds`(3fml) and their FML32 counterparts) has been provided to handle the case of conversion to/from a user type of `FLD_STRING`. 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`.

See Section 3fml of the *BEA TUXEDO Reference Manual* for descriptions of these functions.

Ftypcvt

The functions `CFadd`, `CFchg`, `CFget`, `CFgetalloc`, and `CFfind` use the function `Ftypcvt` to perform the appropriate data conversion. 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)
```

where

- *`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. As an example of its usage, the function `CFchg` is presented:

```
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.

Conversion Rules

A description of conversion rules is now presented. In this description, `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, i.e., any of long, short, float, or double, the conversion is done by the C assignment operator, with proper type casting. For example, converting a short to a float is done by:

  ```
  *((float *)newval) = *((short *) oldval)
  ```
- When converting from a numeric to a string, an appropriate `sprintf` is used. For example, converting a short to a string is done by:

  ```
  sprintf(newval,"%d",*((short *)oldval))
  ```
- When converting from a string to a numeric, the appropriate function (for example, `atof`, `atol`) is used, with the result assigned to a typecasted receiving location, for example:

  ```
  *((float *)newval) = atof(oldval)
  ```
- When converting from type `char` to any numeric type, or from a numeric type to a `char`, the char is considered to be a “shorter short.” For example,

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

  is the method used to convert a `char` to a float. Similarly,

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

  is used to convert a short to a `char`.
- 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` could be used to store a string, less the overhead of the trailing NULL (note that this 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 converting it to a string, and then converting the string 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 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`).

Table 5-2 summarizes the conversion rules presented in this section.

Table 5-2 Summary of Conversion Rules

src typ	**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	-

Table 5-3 defines the entries in Table 5-2.

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

Entry	**Meaning**
-	no conversion need be done, src and dest are same type
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

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)
```

where

- *`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 would be saved by deleting the index.

Findex

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

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

where

- *`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 re-indexing 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 will attempt to drop the entire index if they run out of space before returning a “no space” error.

Frstrindex

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

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

where

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

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)
```

where

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

Example

To transmit a fielded buffer without its index, something similar to the following should be done:

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);
   ```

On the receiving side, the index could be regenerated with 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`; this 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 provide for 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 or from which the 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`, like this:

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

A buffer written with `Fwrite` may be read with `Fread`, as in:

```
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 (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`.

Fchksum

A checksum may be calculated for verifying I/O:

```
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.

Fprint and Ffprint

The function `Fprint` prints a fielded buffer on the standard output in ASCII format:

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

where

- ♦ `fbfr` is a pointer to a fielded buffer

`Ffprint` is similar to `Fprint`, except the text is printed to a specified output stream:

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

where

- ♦ *`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.

Fextread

`Fextread` may be used to construct a fielded buffer from its printed format, i.e. 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 Table 5-4.

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 given, the default action is to `Fadd` the field to the buffer.

Field values may be extended across lines by having the overflow lines begin with a tab (the tab is discarded). A single blank line signals end of buffer; successive blank lines yield a null buffer.

If an error has occurred, `-1` is returned, and `Ferror` is set accordingly. If end of file is reached before a blank line, `Ferror` is set to `FSYNTAX`.

Boolean Expressions of Fielded Buffers

The functions in this section deal with the evaluation of boolean expressions where the "variables" of the expression are the values of fields in a fielded buffer or a VIEW. Functions described in this section allow you to:

- compile 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.

Boolean Expressions

This section describes, in detail, the expressions accepted by the boolean compilation function and how the expression is evaluated. Table 5-5 shows the Backus-Naur Form definitions of the accepted boolean expressions.

Standard C language operators not supported include the shift operators (<< and >>), the bitwise "or" and "and" operators (`||` and &&), the conditional operator (?), the prefix and postfix incrementation and decrementation operators (++ and --), the address and indirection operators (& and *), the assignment operator (=), and the comma operator (,). The following sections describe boolean expressions in greater detail.

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>	?

Field Names and Types

The only variables allowed in the 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 Chapter 3, “Setup,” before using the boolean compilation function. The field types used in booleans are those allowed for FML fields; namely, 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 Chapter 3, “Setup,” before using the boolean compilation function. The field types used in booleans are those allowed for FML VIEWS; namely, 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. NOTE THAT THIS IS NOT AS IT IS IN C where a hexadecimal escape sequence 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 longs and floating point constants are treated as doubles (decimal constants for the `dec_t` type are not supported).

Conversion

To evaluate a boolean expression, the following conversions are performed by the boolean compiler:

- short and int values are converted to longs
- float and decimal values are converted to doubles
- characters are converted to strings
- when comparing a non-quoted string within a field with a numeric, the string is converted to a numeric value
- when comparing a constant (that is, quoted) string with a numeric, the numeric is converted to a string, and a lexical comparison is done
- when comparing a long and a double, the long is converted to a double

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
- `constan` —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 whose 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.

Expression Operators

Table 5-6 lists the precedence of expression operators, with the operators having the highest precedence at the top of the list.

Table 5-6 Boolean Expression Operators

Type	Operators
unary	+, -, !, ~
multiplicative	*, /, %
additive	+, -
relational	< , >, <=, >=, ==, !=
equality and matching	==, !=, %%, !%
exclusive OR	^
logical AND	&&
logical OR	||

Within each operator type, the operators have the same 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

The unary operators recognized are the unary plus operator (+), the unary minus operator (-), the one's complement operator (~), and the logical not operator (!). Expressions with unary operators 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

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

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 strings; 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 strings; if one is a string, it is converted to the arithmetic type of the other.

Equality and Match Operators

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 `recomp`(3c) manual page.

Relational Operators

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

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

```
expression && expression
```

The && operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. 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

The `||` operator groups left-to-right.

```
expression || expression
```

It returns 1 if either of its operands is non-zero, and 0 otherwise. 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
```

Recall that boolean expressions always evaluate to either true or false. The following example:

```
"EMPID[2] %% ’123.*’ && AGE < 32"
```

would be true if field occurrence 2 of `EMPID` exists and begins with the characters “123” and the age field (occurrence 0) appears and is less than 32. This example uses a constant integer as a subscript to `EMPID`. The ? subscript is used in the following example:

```
"PETS[?] == ’dog’"
```

This expression would be true if `PETS` exists and any occurrence of it contained 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)
```

where `*expression` is a pointer to an expression to be compiled.

`Fvboolco` compiles a boolean expression for a VIEW and returns a pointer to an evaluation tree:

```
char *
Fvboolco(char *expression, char *viewname)
```

where `*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,

```
#include "<stdio.h>"
#include "fml.h"
extern char *Fboolco;
char *tree;
. . .
if((tree=Fboolco("FIRSTNAME %% ’J.*n’ && SEX == ’M’")) == NULL)
  F_error("pgm_name");
```

would compile a boolean expression that checks whether the `FIRSTNAME` field is in the buffer, begins with ‘J’ and ends with ‘n’ (e.g., John, Joan, etc.), and whether the `SEX` field is equal to ‘`M`’.

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.

The evaluation tree produced by `Fboolco` is used by the other boolean functions listed below; this avoids having to constantly re-compile the expression.

`free`(3) should be used 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 no longer needed may cause a program to run out of data space.

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)
```

where

- ♦ `*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)
```

where

- ♦ `*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 would yield

```
(((FIRSTNAME[0]) %% ('J.*n')) && ((SEX[0]) == ('M')))
```

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)
```

where

- ♦ *`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:

```
#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");
```

would print "Buffer selected".

`Ffloatev` or `Ffloatev32` is similar to `Fboolev`, but returns the value of the expression as a double. For example,

```
#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));
  }
}
```

would print 6.6. If `Fboolev` were used in place of `Ffloatev` in the above example, a 1 would be printed.

VIEW Conversion to and from Target Format

A VIEW can be converted to and from a target record format. The default target format is IBM System/370 COBOL records.

Fvstot, Fvttos and Fcodeset

The three functions that provide target conversion are as follows.

```
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 from an FML buffer to a target record, first call `Fvftos` to convert the FML buffer to a C structure, and call `Fvstot` to convert to a target record. To convert from a target record to an FML buffer, first call `Fvttos` to convert to a C structure and then call `Fvstof` to convert the structure to an FML buffer.

The default target is IBM/370 COBOL records. The default data conversion is done as shown in Table 5-7.

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 the 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 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/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 that the decimal occupies. The number to the right is the number of digits to the right of the decimal point. The formula for conversion is:

```
dec_t(m, n) <=> S9(2*m-(n+1))V9(n)COMP-3
```

Decimal values may be converted to and from other data types (e.g., int, long, string, double, and float) using the functions described in `decimal`(3c).

See the `Fvstof`(3fml) reference page for the default character conversion of ASCII to/from EBCDIC.

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.

6 Examples

VIEWS Examples

The VIEWS examples that follow are unrelated to the example FML program that appears later in this chapter.

Sample Viewfile

The following 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 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 figure 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(1)

The following is a header file produced from a field table file by `mkfldhdr`(1); assume that a field table file containing the field definitions of the fields shown in the previous examples was used as input to `mkfldhdr`(1).

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 is the COBOL COPY file, `CUSTDB.cbl`, produced by `viewc` with the `-C` command line 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).
```

A sample COBOL program including a COBOL COPY file produced by `viewc -C` is shown in the *BEA TUXEDO COBOL Guide*.

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 Chapter 3, “Setup,” must be properly set for this program to work.

Information on compiling FML programs can be found on the `compilation`(5) reference page in the *BEA TUXEDO Reference Manual*.

Listing 6-6 Sample VIEWS Program

```
/* sample VIEWS program */
#include stdio.h>
#include "fml.h"
#include "custdb.flds.h"     /* field header file shown in Fig. 6-3 */
#include "custdb.h"          /* C structure header file produced by */
/* viewc shown in Fig. 6.2    */
#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.

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 BEA TUXEDO system data entry masks under the control of `mio`. It is especially worth noting that in these servers the ATMI functions `tpalloc`() and `tprealloc`() are used to allocate message buffers, rather than the FML functions `Falloc`() and `Frealloc`().

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
`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

Table A-1 FML Error Codes and Messages

Error Code	#	Error Message
`FVFSYNTAX`	16	syntax error in viewfile
`FVFOPEN`	17	cannot find or open viewfile
`FBADACM`	18	ACM contains negative value
`FNOCNAME`	19	cname not found