BEA MessageQ

FML Programmer's Guide

BEA MessageQ Version 5.0
Document Edition 5.0
October 1998

BEA MessageQ FML Programmer’s Guide

Document Edition	Date	Software Version
Version 5.0	October 1998	BEA MessageQ, Version 5.0

Contents

CHAPTER

1 Introduction

About This Guide and FML

This chapter describes the contents of the guide, how the Field Manipulation Language (FML) fits into the BEA MessageQ system, and how you might get the most out of the guide. We assume that you are familiar with the BEA MessageQ system.

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.

The original FML allowed for 16-bit field identifiers, field lengths, field occurrences, and record lengths. A newer FML32 interface allows for larger identifiers (32-bit), field lengths, field occurrences, and record lengths. The interfaces are nearly identical; the only difference is that a suffix of "32" is added to the name of type definitions, header files, functions, and commands.

Note: BEA MessageQ only supports FML32. Do not use the 16-bit FML functions in developing MessageQ applications.

How Does FML32 Fit into the BEA MessageQ System?

Within the BEA MessageQ system, FML32 functions are used to manipulate fielded buffers. In MessageQ applications, messages may be sent as message buffers (predefined, static data structures) or as FML32 buffers. Using FML32, applications construct messages containing both the message content and the information needed by the receiver program to understand what is in the message.

Who Is This Document For?

This guide gives detailed information about the features of FML32 and how the different FML32 functions are used.

This guide is intended for programmers who need to learn how to use FML32 functions in programming BEA MessageQ applications. This guide also provides information for users of applications that make use of FML32 with regard to setting up the environment correctly.

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.
- The BEA MessageQ system—We assume, even if you have not yet worked on a BEA MessageQ application, that you at least have an understanding of what the BEA MessageQ system is intended to do, and that you have read about the application development environment in the *BEA MessageQ Programmer's Guide*.

What Does This Document Include?

- **Concepts and Definitions**—Several definitions are provided to explain ideas and terms that are used in the guide.
- **An Overview of FML32**—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 MessageQ system in general, and FML32 in particular. This chapter also shows you how to customize your installed FML32 software.
- **Defining and Using FML32 Fielded Buffers**—Chapter 4 outlines the use of the FML32 software, and how to set up your C language programs to use the software.
- **FML32 Field Manipulation Functions**—Chapter 5 deals with how to use the FML32 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. Chapter 6 has provides additional examples.

What Other FML32 Documentation Is There?

In addition to this guide, documentation on FML32 function calls can be found in the reference page for each FML32 function and the following related reference pages in the *BEA MessageQ Reference Manual*:

Table 1-1 Section 5 reference pages

Reference Page	Description
`field_tables`(5)	describes the structure of FML field tables
`mkfldhdr32`	describes the command used to create header files from field tables

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).

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 FML32 records. It then describes the features of the Field Manipulation Language and the circumstances under which you might want to use them.

A comparison of FML32 records with traditional structured records clearly shows the advantages of using fielded buffers throughout an application.

Dividing Records into Fields

Unless a data record is a complete and indivisible entity (an unusual situation), you need to be able to break a record into fields so you can use or change the information in the record. In BEA MessageQ applications there are two ways to divide records into fields:

- ♦ Through message buffers (predefined C language data structures)
- ♦ 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, a data item in a C language program that contains information about an employee's identification number, name, address, and sex, may be formatted in a structure such as the following:

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

where the data type of the `empid` field 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.

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; as a result, space if often wasted. (Not all fields will always contain a value and 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, a field is referenced by an identifier rather than by the fixed offset prescribed by record structures, and all access to fields is through function calls.

Fielded buffers can be used throughout a BEA MessageQ application as the standard method of representing data sent between cooperating processes.

Implementing Fielded Buffers with FML32

Fielded buffers are created, updated, accessed, input, and output via the Field Manipulation Language (FML). FML32 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.

FML32 is implemented as a library of functions and macros that can be called from C programs. There are two major groups of FML32 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.

FML32 Features

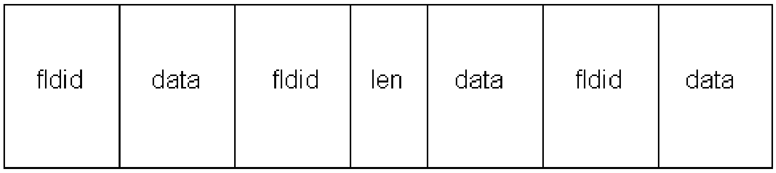
This section describes the features of FML32 and recommends how to use them in application programs.

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 FML32 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 `fldid32`. For variable-length items, fldid32 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 FML32 functions are used (fml32.h), field identifiers are typedef'd as FLDID32, field value lengths as FLDLEN32, and field occurrence numbers as FLDOCC32.

Supported Field Types

The supported field types are short, long, float, double, character, string, and carray (character array). These types are #define'd in fml32.h as shown in Listing 2-1.

Listing 2-1 FML32 field types as defined in 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 ways:

- 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 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 FML32 conversion functions).

Most FML32 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 FML32 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.

Field Name to Identifier Mappings

A field is usually referred to by its field identifier (`fldid32`), an integer. (See Chapter 4, “Field Definition and Use,” 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 ASCII 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 FML32 to access the data in fielded records, there must be some way for FML32 to access the field name/identifier mappings. FML32 gets this information in one of two ways:

- At run-time, through UNIX field table files and FML32 mapping functions
- At compile-time, through C header files

Run-Time: Field Table Files

Field name/identifier mappings can be made available to FML32 programs at run-time through field table files. It is the responsibility of the programmer to set two environment variables that tell FML32 where the field name/identifier mapping table files are located.

The environment variable `FLDTBLDIR32` contains a list of directories where field tables can be found. The environment variable `FIELDTBLS32` contains a list of the files in the table directories that are to be used.

Within application programs, the FML32 function `Fldid32()` provides for a run-time translation of a field name to its field identifier. `Fname32()` translates a field identifier to its field name (see `Fldid`(3fml) and `Fname`(3fml)). 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

`mkfldhdr32`(1) 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 `Fldid32()` or `Fname32()` functions as you would with the field table files described in the previous section.

If you always know the field names needed by your program, you can `#include` your field table header file(s), saving some data space and enabling your program to run 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.

Fielded Buffer Indexes

When a fielded buffer has many fields, access is expedited in FML32 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.

FML32 provides the `Funidex32`() function for discarding the index. When the fielded buffer is read from disk (or received from a sending process), the index can be explicitly reconstructed with the function `Findex32`().

Note that these space savings do not apply to memory. The function `Funidex32`() does not recover in-core memory used by the index of a fielded buffer.

Multiple Occurrences of Fields

A fielded buffer may contain more than one occurrence of any field. Many FML32 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 constitutes 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 determine the next action taken by the application program. FML32 provides several functions that create boolean expressions on fielded buffers and determine if a given buffer 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 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 FML32 function is used to update another buffer, Buffer B, with data from Buffer A.

Error Handling

When an FML32 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 `FLDID32`
- ♦ -1 is returned for all others

All FML32 function call returns should be checked against the appropriate value above to detect errors.

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

The `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 `Ferror32`, which is set when errors occur.

To be most useful, the argument string to the `F_error32`() function should include the name of the program that incurred the error.

`Fstrerror32`() 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 `F_error32`().

The error codes that can be produced by an FML32 function are described on the page that documents the function in the *BEA MessageQ Reference Manual*.

3 Setup

Introduction

This chapter deals with the setup of the FML32 environment. Before you can begin to work with FML32 fielded buffers you must set environment variables appropriate for your application. These activities are described in this chapter.

Directory Structure

The delivered FML32 software will reside in a subtree of the local file system. Several of the FML32 modules assume that the structure of this subtree is as described in this section. The sub-directories are:

- `include`—contains header files needed by writers of C application code.
- `bin`—contains the executable commands of FML.
- `lib`—contains subroutine packages of FML; when compiling a program that uses FML32 functions, `$MESSAGEQ/lib/libfml32.`*`suffix`* and `$MESSAGEQ/lib/libgp.`*`suffix`* should be included on the C compiler command line to resolve external references. (The suffix is `.a` for POSIX operating systems without shared objects, `.so` for use of shared objects, and `.lib` for Windows 95 and Windows NT.)

C application software using FML32 must include the following header files in the order shown:

```
#include <stdio.h>
#include “fml32.h”
```

Environment Variables

This section describes several environment variables used by FML32.

The following variable is used in FML32 to search for system supplied files:

- `TUXDIR`—this variable should be set to the topmost node of the installed BEA MessageQ system software including FML32.

The following variables are used throughout FML32 to access field table files (described in Chapter 4):

- `FIELDTBLS32`—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 `FLDTBLDIR32` variable. If `FIELDTBLS32` is not set, then the single file name `fld.tbl` is used (`FLDTBLDIR32` still applies; see below.)
- `FLDTBLDIR32`—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 `FLDTBLDIR32` is not set or is null, then its value is assumed to be the current directory.

4 Field Definition and Use

Introduction

Before you can begin to work with FML32 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)

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 using `typedef` as a `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 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 FML32 is used with other software that also uses fields, additional restrictions may be imposed on field numbers.

The numbering convention adopted by the BEA MessageQ is as follows:

- field numbers 1-100 are reserved for system use
- field numbers 101-33,554,431 are for application-defined fields with 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 `FLDTBLDIR32` and `FIELDTBLS32` to specify the source directories and field table files, respectively, which are to be used. These should be set as described in Chapter 3.

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 `mkfldhdr32`(1); for example, this

would allow the application to pass C comments, `what` strings, etc. to the generated C header file.

- 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: `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 `Fldid32`() and `Fname32`() functions that consult the set of field table files specified by the `FLDTBLDIR32` and `FIELDTBLS32` environment variables.

`Fldid32`() maps its argument, a field name, to a `fieldid`:

```
char *name;
extern FLDID32 Fldid32();
FLDID32 id;
...
id = Fldid32(name);
```

`Fname`() does the reverse translation by mapping its argument, a `fieldid`, to a field name:

```
extern char *Fname32();
name = Fname32(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, `Fldid32()` loads the field table files and performs the required search. Thereafter, the files are kept loaded. `Fldid32()` returns the field identifier corresponding to its argument on success, and returns `BADFLDID` on failure, with `Ferror32` set to `FBADNAME`.

To recover the data space used by the field tables loaded by `Fldid32()`, the user may unload all of the files by a call to the `Fnmid_unload32()` function.

The function `Fname32()` acts in a fashion similar to `Fldid32`(), 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, `Fname32()` returns a pointer to a character string containing the name corresponding to the `fldid` argument. On failure, `Fname32()` returns NULL.

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

As with `Fldid32()`, 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 `Fname32()` may be recovered by a call to the function `Fidnm_unload32()`.

Both mapping functions and other FML32 functions that use run-time mapping require `FIELDTBLS32` and `FLDTBLDIR32` to be set properly. Otherwise, default values are used (see Chapter 3 for the defaults).

Field Header Files

The command `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).

mkfldhdr32(1) translates each field-table specified in the FIELDTBLS32 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 mkfldhdr32(1) by specifying a -d option followed by the name of the directory in which you want the header files to reside. For example,

```
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 ${FLDTBLDIR32}/maskftbl, ${FLDTBLDIR32}/DBftbl and ${FLDTBLDIR32}/miscftbl. The command

```
mkfldhdr32 -d${FLDTBLDIR32}
```

will process the sample input field-table files and produce the same output files, but will place them in the directory given by ${FLDTBLDIR32}.

You may override the environment variables (or avoid setting them) when using mkfldhdr32 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, FLDTBLDIR32 is assumed to be the current directory and FIELDTBLS32 is assumed to be the list of parameters that the user specified on the command line. For example,

```
mkfldhdr32 myfields
```

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

CHAPTER

5 Field Manipulation Functions

Introduction

This chapter describes all FML32 functions exception run-time mapping (which is described in Chapter 4). In this chapter you will learn:

- FML32 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

FML and FML32

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.

A second interface, FML32, uses 32-bit values for field lengths and identifiers. It allows for about 30 million fields, and field and buffer lengths of about 2 billion bytes. The definitions, types, and function prototypes for FML32 are in `fml32.h`. Functions live in `-lfml32`.

BEA MessageQ supports only FML32. Do not use 16-bit FML functions in developing MessageQ applications.

The names of all definitions, types, and functions for FML32 have a “`32`” suffix (for example, `MAXFBLEN32`, `FBFR32`, `FLDID32`, `FLDLEN32`, `Fchg32`(), and error code `Ferror32`()). Also the environment variables are suffixed with “`32`” (for example, `FLDTBLDIR32` and `FIELDTBLS32`). 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.

FML32 Parameters

To make it easier to remember the parameters for the FML32 functions, a convention has been adopted for the sequence of function parameters. FML32 parameters appear in the following sequence:

1. For functions that require a pointer to a fielded buffer (`FBFR32`), 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 `Ferror32`() set to `FALIGNERR`) and the area must be a fielded buffer (or an error is returned with `Ferror32`() 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 `FLDID32`) appears next (in the case of `Fnext32`(), it is a pointer to a field identifier).
4. For functions that need a field occurrence (type `FLDOCC32`), this parameter comes next (for `Fnext32`(), 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 `FLDLEN32`). 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.

Fldid32

`Fldid32`() 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:

```
FLDID32
Fldid32(char *name)
```

where *name* is a valid field name.

The space used by the mapping tables in memory can be freed using the `Fnmid_unload32`() function. Note that these tables are separate from the tables loaded and used by the `Fname32`() function.

Fname32

`Fname32`() 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 *
Fname32(FLDID32 fieldid)
```

where *fieldid* is a valid field identifier.

The space used by the mapping tables in memory can be freed using the `Fidnm_unload32`() function. Note that these tables are separate from the tables loaded and used by the `Fldid32`() function.

Fldno32

`Fldno32`() extracts the field number from a given field identifier:

```
FLDOCC32
Fldno32(FLDID32 fieldid)
```

where *fieldid* is a valid field identifier.

Fldtype32

`Fldtype32`() extracts the field type (an integer, as defined in `fml32.h`) from a given field identifier.

```
int
Fldtype32(FLDID32 fieldid)
```

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

Table 5-1 shows the possible values returned by `Fldtype32`() 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

Ftype32

`Ftype32`() returns a pointer to a string containing the name of the type of a field given a field identifier:

```
char *
Ftype32(FLDID32 fieldid)
```

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

For example:

```
char *typename
. . .
typename = Ftype32(fieldid);
```

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

Fmkfldid32

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. `Fmkfldid32`() provides this functionality:

```
FLDID32
Fmkfldid32(int type, FLDID32 num)
```

where

- ♦ *`type`* is a valid type (an integer; see `Fldtype32`(), above)
- ♦ *`num`* is a field number (it should be an unused field number, to avoid confusion with existing fields)

Buffer Allocation and Initialization

Most FML32 functions require a pointer to a fielded buffer as an argument. The typedef `FBFR32` is available for declaring such pointers, as in this example:

```
FBFR32 *fbfr32;
```

In this chapter, the variable *`fbfr32`* 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.

Fielded32

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

```
int
Fielded32(FBFR32 *fbfr32)
```

`Fielded32`() returns true (1) if the buffer is fielded. If the buffer is not fielded, `Fielded32`() returns false (0) and does not set `Ferror32`().

Fneeded32

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

```
long
Fneeded32(FLDOCC32 F, FLDLEN32 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 (for example, a long is stored as a long and needs four bytes). For a variable length field, you should estimate the average amount of space needed. The space calculated by `Fneeded` includes a fixed overhead for each field in addition to the space needed for the field values.

Once you obtain the estimate of space from `Fneeded32`(), 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((fbfr32 = (FBFR32 *)malloc(Fneeded32(NF, NV))) == NULL)
      F_error("pgm_name");   /* no space to allocate buffer */
```

However, this allocated memory space is not yet a fielded buffer. `Finit32`() must be used to initialize it.

Finit32

The `Finit32`() function initializes an allocated memory space as a fielded buffer.

```
int
Finit32(FBFR32 *fbfr32, FLDLEN32 buflen)
```

where

- ♦ *`fbfr32`* is a pointer to an uninitialized fielded buffer
- ♦ *`buflen`* is the length of the buffer, in bytes

A call to `Finit32`() to initialize the memory space allocated in the example above (in the `Fneeded32`() section) would look like the following:

```
Finit32(fbfr32, Fneeded32(NF, NV));
```

Now `fbfr32` points to an initialized, empty fielded buffer. Up to `Fneeded32(NF, NV)` bytes minus a small amount are available in the buffer to hold fields.

Note: The numbers used in the `malloc`(3) call (see the example in the `Fneeded32`() section) and `Finit32`() call must be the same.

Falloc32

Calls to `Fneeded32`(), `malloc`(3) and `Finit32`() may be replaced by a single call to `Falloc32`(), which allocates the desired amount of space and initializes the buffer.

```
FBFR32 *
Falloc32(FLDOCC32 F, FLDLEN32 V)
```

where

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

A call to `Falloc32`() that would replace the examples above would look like the following:

```
extern FBFR32 *Falloc32;
. . .
```

```
 if((fbfr32 = Falloc32(NF, NV)) == NULL)
      F_error(“pgm_name”);   /* couldn't allocate buffer */
```

Storage allocated with `Falloc32`() (or `Fneeded32`(), `malloc`(3) and `Finit32`()) should be freed with `Ffree32`().

Ffree32

`Ffree32`() is used to free memory space allocated as a fielded buffer.

```
int
Ffree32(FBFR32 *fbfr32)
```

where *`fbfr32`* is a pointer to a fielded buffer

For example:

```
 #include  <fml32.h>
 . . .
 if(Ffree32(fbfr32)  0)
       F_error("pgm_name");     /* not fielded buffer */
```

`Ffree32`() is recommended as opposed to `free`(3), because `Ffree32`() 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 following code is analogous to the preceding example but `Fneeded32`() 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)];
FBFR32 *fbfr32=(FBFR32 *)buffer;
. . .
Finit32(fbfr32, 500);
```

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

```
FBFR32 badfbfr;
. . .
Finit32(&badfbfr, Fneeded32(NF, NV));
```

The structure for `FBFR32` is not defined in the user header files so this code will produce a compilation error.

Fsizeof32

`Fsizeof32`() returns the size of a fielded buffer in bytes:

```
long
Fsizeof32(FBFR32 *fbfr32)
```

where *`fbfr32`* is a pointer to a fielded buffer

For example:

```
long bytes;
. . .
bytes = Fsizeof32(fbfr32);
```

`Fsizeof32`() returns the same number that `Fneeded32`() returned when the fielded buffer was originally allocated.

Funused32

`Funused32`() may be used to determine how much space is available in a fielded buffer for additional data:

```
long
Funused32(FBFR32 *fbfr32)
```

where *`fbfr32`* is a pointer to a fielded buffer

For example:

```
long unused;
. . .
unused = Funused32(fbfr32);
```

Note that `Funused32`() does not indicate where, in the buffer, the unused bytes are located; it indicates only the number of unused bytes.

Fused32

`Fused32`() may be used to determine how much space is used in a fielded buffer for data and overhead:

```
long
Fused32(FBFR32 *fbfr32)
```

where *`fbfr32`* is a pointer to a fielded buffer

For example:

```
long used;
. . .
used = Fused32(fbfr32);
```

Note that `Fused32`() does not indicate where, in the buffer, the used bytes are located; it indicates only the number of used bytes.

Frealloc32

At some point (such as during the addition of a new field value) the buffer may run out of space. `Frealloc32`() can be used to increase (or decrease) the size of the buffer:

```
FBFR32 *
Frealloc32(FBFR32 *fbfr32, FLDOCC32 nf, FLDLEN32 nv)
```

where

- *`fbfr32`* 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:

```
FBFR32 *newfbfr32;
. . .
if((newfbfr32 = Frealloc32(fbfr32, NF+5, NV+300)) == NULL)
       F_error32(“pgm_name”);     /* couldn't re-allocate space */
else
       fbfr32 = newfbfr32;          /* assign new pointer to old */
```

In this case, the application needed to remember the number of fields and the number of bytes of space previously allocated for field values. Note that the arguments to `Frealloc32`() (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
FBFR32 *newfbfr32;
. . .
if((newfbfr32 = Frealloc32(fbfr32, 0, Fsizeof(fbfr32)+INCR)) ==
NULL)
      F_error32("pgm_name");       /* couldn't re-allocate space */
else
      fbfr32 = newfbfr32;           /* assign new pointer to old */
```

Note that you do not need to know the number of fields or the amount of space for field values 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 space for field values. The above example could be executed as many times as needed without remembering past executions or values. The user need not call `Finit32`() after calling `Frealloc32`().

If the amount of additional space requested in the call to `Frealloc32`() is contiguous to the old buffer, `newfbfr32` and `fbfr32` in the examples above will be the same. However, defensive programming dictates that the user should declare `newfbfr32` as a safeguard against the case where either a new value or NULL is returned. If `Frealloc32`() fails, do not use `fbfr32` again.

Note: You cannot reduce the size of a fielded buffer to less than the amount of space (in 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.

Fmove32

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:

```
FBFR32 *src;
char *dest;
. . .
memcpy(dest, src, Fsizeof32(src));
```

The function `memcpy` (one 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.

`Fmove32`() acts like `memcpy`, but does not need an explicit length (it is computed):

```
int
Fmove32(char *dest, FBFR32 *src)
```

where

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

For example:

```
FBFR32 *src;
char *dest;
. . .
if(Fmove32(dest,src) < 0)
        F_error("pgm_name");
```

`Fmove32`() 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 `Falloc32`()), but it must be aligned on a `short` boundary (4-byte alignment for FML32). Thus, `Fmove32`() provides an alternative to `Fcpy32`() (see below) when it is desired to copy a fielded buffer to a non-fielded buffer, but `Fmove32()` does not check to make sure there is enough room in the destination buffer to receive the source buffer.

Fcpy32

`Fcpy32()` is used to overwrite one fielded buffer with another:

```
int
Fcpy32(FBFR32 *dest, FBFR32 *src)
```

where

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

`Fcpy32()` preserves the overall buffer length of the overwritten fielded buffer; thus, `Fcpy32()` is useful for expanding or reducing the size of a fielded buffer. For example:

```
FBFR32 *src, *dest;
. . .
if(Fcpy32(dest, src) 0)
        F_error32("pgm_name");
```

Unlike `Fmove32()`, where *`dest`* could point to an uninitialized area, `Fcpy32()` expects *`dest`* to point to an initialized fielded buffer (allocated using `Falloc32()`) 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 to less than the amount of space (in bytes) currently being used in the buffer.

As with `Fmove32()`, the source buffer is not modified by `Fcpy32()`.

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.

Fadd32

The `Fadd32`() function adds a new field value to the fielded buffer.

```
int
Fadd32(FBFR32 *fbfr32, FLDID32 fieldid, char *value, FLDLEN32 len)
```

where

- *`fbfr32`* 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, `Fchg32`() must be used.)

`Fadd32`(), 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 `FLDLEN32`. For example:

```
FLDID32 fieldid, Fldid32;
FBFR32 *fbfr32;
```

```
. . .
fieldid = Fldid32("fieldname");
if(Fadd32(fbfr32, fieldid, "new value", (FLDLEN32)9) < 0)
          F_error32("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(Fadd32(fbfr32, fieldid, lval, (FLDLEN32)0) < 0)
          F_error32("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`).

Fappend32

The `Fappend32()` function appends a new field value to the fielded buffer.

```
int
Fappend32(FBFR32 *fbfr32, FLDID32 fieldid, char *value, FLDLEN32
len)
```

where

- ♦ *`fbfr32`* 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`

`Fappend32`() 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 FML32 routines are allowed in append mode: `Fappend32`(), `Findex32`(), `Funindex32`(), `Ffree32`(), `Fused32`(), `Funused32`() and `Fsizeof32`(). Calls to `Findex32`() or `Funindex32`() will end append mode. The following example shows the construction of a 500-row buffer with five fields per row using `Fappend32`().

```
for (i=0; i 500 ;i++) {
   if ((Fappend32(fbfr32, LONGFLD1, &lval1[i], (FLDLEN32)0) < 0) ||
       (Fappend32(fbfr32, LONGFLD2, &lval2[i], (FLDLEN32)0) < 0) ||
       (Fappend32(fbfr32, STRFLD1, &str1[i], (FLDLEN32)0) < 0) ||
       (Fappend32(fbfr32, STRFLD2, &str2[i], (FLDLEN32)0) < 0) ||
       (Fappend32(fbfr32, LONGFLD3, &lval3[i], (FLDLEN32)0) < 0)) {
           F_error32("pgm_name");
        break;
  }
}
Findex32(fbfr32, 0);
```

`Fappend32`(), 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 `FLDLEN32`.

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`).

Fchg32

`Fchg32`() changes the value of a field in the buffer.

```
int
Fchg32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc, char *value,
FLDLEN32 len)
```

where

- ♦ *`fbfr32`* 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 `Fadd32`())
- ♦ *`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`:

```
FBFR32 *fbfr32;
FLDID32 fieldid;
FLDOCC32 oc;
FLDLEN32 len;
char value[50];
. . .
strcpy(value, "new value");
flen = strlen(value);
if(Fchg32(fbfr32, fieldid, oc, value, len) < 0)
       F_error32("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`).

Fcmp32

`Fcmp32`() compares the field identifiers and field values of two fielded buffers.

```
int
Fcmp32(FBFR32 *fbfr321, FBFR32 *fbfr322)
```

where

- ♦ *`fbfr321`* and *`fbfr322`* 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 *`fbfr321`* field is less than the field id of the corresponding field of *`fbfr322`*
- ♦ the value of a *`fbfr321`* field is less than the value of the corresponding field of *`fbfr322`*
- ♦ *`fbfr1`* is shorter than *`fbfr322`*

`Fcmp32`() returns a 1 if the reverse of any of the above conditions is true (for example, if the field ID of a *`fbfr322`* field is less than the field ID of the corresponding field of *`fbfr321`*, and so on).

Fdel32

The `Fdel32`() function deletes the specified field occurrence.

```
int
Fdel32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc)
```

where

- ♦ *fbfr32* is a pointer to a fielded buffer
- ♦ *fieldid* is a field identifier
- ♦ *oc* is the occurrence number

For example,

```
FLDOCC32 occurrence;
. . .
occurrence=0;
if(Fdel32(fbfr32, fieldid, occurrence) < 0)
            F_error32("pgm_name");
```

deletes the first occurrence of the field indicated by the specified field identifier. If it does not exist, the function returns `-1` (`Ferror32`() is set to `FNOTPRES`).

Fdelall32

`Fdelall32`() deletes all occurrences of the specified field from the buffer:

```
int
Fdelall32(FBFR32 *fbfr32, FLDID32 fieldid)
```

where

- ♦ *fbfr32* is a pointer to a fielded buffer
- ♦ *fieldid* is a field identifier

For example:

```
if(Fdelall32(fbfr32, fieldid) < 0)
     F_error32("pgm_name");        /* field not present */
```

If the field is not found, the function returns `-1` (`Ferror32`() is set to `FNOTPRES`).

Fdelete32

`Fdelete32`() deletes all occurrences of all fields listed in the array of field identifiers, `fieldid[]`:

```
int
Fdelete32(FBFR32 *fbfr32, FLDID32 *fieldid)
```

where

- ♦ *`fbfr32`* 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 "fld.tbl.h"
FBFR32 *dest;
FLDID32 fieldid[20];
. . .
fieldid[0] = A;   /* field id for field A */
fieldid[1] = D;   /* field id for field D */
fieldid[2] = BADFLDID;   /* sentinel value */
if(Fdelete32(dest, fieldid) < 0)
         F_error32("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.

`Fdelete32`() is a more efficient way of deleting several fields from a buffer than using several `Fdelall32`() calls.

Ffind32

`Ffind32`() finds the value of the specified field occurrence in the buffer:

```
char *
Ffind32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc, FLDLEN32
*len)
```

where

- *`fbfr32`* 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 `Ffind32`() 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 "fld.tbl.h"
FBFR32 *fbfr32;
FLDLEN32 len;
char* Ffind32, *value;
. . .
if((value=Ffind32(fbfr32,ZIP,0, &len)) == NULL)
      F_error32("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 `Ferror32`() is set to `FNOTPRES`.

`Ffind32`() is useful for gaining "read-only" access to a field. The value returned by `Ffind32`() should not be used to modify the buffer. Field value modification should be done only by the function `Fadd32`() or `Fchg32`().

The value returned by `Ffind32`() 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;
FLDLEN32 length;
char *Ffind32;
. . .
if((l1=(long *)Ffind32(fbfr32, ZIP, 0, &length)) == NULL)
        F_error32("pgm_name");
else
        l2 = *l1;
```

and should be re-written as:

```
if((l1==(long *)Ffind32(fbfr32, ZIP, 0, &length)) == NULL)
        F_error32("pgm_name");
else
        memcpy(&l2,l1,sizeof(long));
```

Ffindlast32

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 *
Ffindlast32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 *oc, FLDLEN32
*len)
```

where

- `fbfr32` 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.

`Ffindlast32`() acts like `Ffind32`(), 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 `Ffindlast32`() is valid only as long as the buffer remains unchanged.

Ffindocc32

`Ffindocc32`() 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:

```
FLDOCC32
Ffindocc32(FBFR32 *fbfr32, FLDID32 fieldid, char *value, FLDLEN32
len;)
```

where

- *fbfr32* 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 `Fadd32`())
- *len* is the length of the value if type `carray`

For example,

```
#include "fld.tbl.h"
FBFR32 *fbfr32;
FLDOCC32 oc;
long zipvalue;
. . .
zipvalue = 123456;
if((oc=Ffindocc32(fbfr32,ZIP,&zipvalue, 0)) < 0)
       F_error32("pgm_name");
```

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

Regular expressions are supported for string fields. For example,

```
#include "fld.tbl.h"
FBFR32 *fbfr32;
FLDOCC32 oc;
char *name;
. . .
name = "J.*"
if ((oc = Ffindocc32(fbfr32, 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 `Ffindocc32`() must be nonzero. If it is zero, a simple string compare is performed. If the field value is not found, `-1` is returned.

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

Fget32

`Fget32`() should be used to retrieve a field from a fielded buffer when the value is to be modified:

```
int
Fget32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc, char *loc, FLDLEN32 *maxlen)
```

where

- ♦ `fbfr32` 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 `Fget32` 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.

`Fget32`() returns an error if the desired field is not in the buffer (`FNOTPRES`), or if the destination buffer is too small (`FNOSPACE`). For example,

```
FLDLEN32 len;
char value[100];
. . .
len=sizeof(value);
if(Fget32(fbfr32, ZIP, 0, value, &len) < 0)
          F_error32("pgm_name");
```

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

```
FLDLEN32 len;
long value;
. . .
len = sizeof(value);
if(Fget32(fbfr32, ZIP, 0, value, &len) < 0)
          F_error32("pgm_name");
```

Fgetalloc32

Like `Fget32`(), `Fgetalloc32`() finds and makes a copy of a buffer field, but it acquires space for the field via a call to `malloc`(3):

```
char *
Fgetalloc32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc, FLDLEN32
*extralen)
```

where

- ♦ *`fbfr32`* 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 `Fgetalloc32`() is shown as a character pointer data type (`char*` in C). The actual type of the pointer returned is the same as the type of the value to which it points.

On success, `Fgetalloc32`() returns a valid pointer to the copy of the properly aligned buffer field; on error it returns NULL. If `malloc`(3) fails, `Fgetalloc32`() returns an error (`Ferror32`() is set to `FMALLOC`).

The last parameter to `Fgetalloc32`() 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:

```
FLDLEN32 extralen;
FBFR32 *fieldbfr
char *Fgetalloc32;
. . .
```

```
extralen = 0;
if (fieldbfr = (FBFR32 *)Fgetalloc32(fbfr32, ZIP, 0, &extralen) ==
NULL)
        F_error32("pgm_name");
```

It is the responsibility of the caller to `free` space acquired by `Fgetalloc32`().

Fgetlast32

`Fgetlast32`() is used to retrieve the last occurrence of a field from a fielded buffer when the value is to be modified:

```
int
Fgetlast32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 *oc, char *loc, FLDLEN32
*maxlen)
```

where

- ♦ *`fbfr32`* 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 `Fgetlast32`() with a pointer to a private buffer, as well as the length of the buffer. `Fgetlast32`() acts like `Fget32`(), 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.

Fnext32

`Fnext32`() finds the next field in the buffer after the specified field occurrence:

```
int
Fnext32(FBFR32 *fbfr32, FLDID32 *fieldid, FLDOCC32 *oc, char *value, FLDLEN32
*len)
```

where

- *fbfr32* 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 (`Ferror32`() 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, `Fnext32`() 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:

```
FLDID32 fieldid;
FLDOCC32 occurrence;
char *value[100];
FLDLEN32 len;
. . .
for(fieldid=FIRSTFLDID,len=sizeof(value);
     Fnext32(fbfr32,fieldid,&occurrence,value,&len) > 0;
     len=sizeof(value)) {
   /* code for each field occurrence */
}
```

Fnum32

`Fnum32`() returns the number of fields contained in the specified buffer, or `-1` on error:

```
FLDOCC32
Fnum(FBFR32 *fbfr32)
```

where

- ♦ *`fbfr32 is a`* pointer to a fielded buffer

For example:

```
if((cnt=Fnum32(fbfr32)) < 0)
  F_error32("pgm_name");
else
  fprintf(stdout,"%d fields in buffer\n",cnt);
```

would print the number of fields in the specified buffer.

Foccur32

`Foccur32`() returns the number of occurrences for the specified field in the buffer:

```
FLDOCC32
Foccur32(FBFR32 *fbfr32, FLDID32 fieldid)
```

where

- ♦ *`fbfr32`* 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:

```
FLDOCC32 cnt;
. . .
if((cnt=Foccur32(fbfr32,ZIP)) < 0)
 F_error32("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.

Fpres32

`Fpres32`() returns true (1) if the specified field occurrence exists and false (0) otherwise:

```
int
Fpres32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc)
```

where

- *fbfr32* is a pointer to a fielded buffer
- *fieldid* is a field identifier
- *oc* is the occurrence number

For example:

```
Fpres32(fbfr32,ZIP,0)
```

would return true if the field `ZIP` exists in the fielded buffer pointed to by `fbfr32`.

Fvals32 and Fvall32

`Fvals32`() works like `Ffind32`() for `string` values but guarantees that a pointer to a value is returned. `Fvall32`() works like `Ffind32`() for `long` and `short` values, but returns the actual value of the field as a `long`, instead of a pointer to the value.

```
char*
Fvals32(FBFR32 *fbfr32,FLDID32 fieldid,FLDOCC32 oc)


char*
Fvall32(FBFR32 *fbfr32,FLDID32 fieldid,FLDOCC32 oc)
```

where in both functions

- *fbfr32* is a pointer to a fielded buffer
- *fieldid* is a field identifier
- *oc* is the occurrence number

For `Fvals32`(), 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 `Fvall32`(), if the specified field occurrence is not found, then 0 is returned. This function is useful for passing the value of a field to another function without checking the return value. This function is valid only for fields of type `long` and `short`; 0 is automatically returned for other field types (that is, no conversion is done).

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

Fconcat32

`Fconcat32`() adds fields from the source buffer to the fields that already exist in the destination buffer.

```
int
Fconcat32(FBFR32 *dest, FBFR32 *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:

```
FBFR32 *src, *dest;
. . .
if(Fconcat32(dest,src) < 0)
       F_error32("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.

Fjoin32

`Fjoin32`() is used to join two fielded buffers based on matching fieldid/occurrence.

```
int
Fjoin32(FBFR32 *dest, FBFR32 *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(Fjoin32(dest,src) < 0)
     F_error32("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 `Frealloc32()`) and the `Fjoin32`() function may be repeated. (Even if the destination buffer has been partially updated, repeating the function will give the correct results.)

Fojoin32

Fojoin32() is similar to Fjoin32(), but it does not delete fields from the destination buffer that have no corresponding fieldid/occurrence in the source buffer.

```
int
Fojoin32(FBFR32 *dest, FBFR32 *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(Fojoin32(dest,src) < 0)
      F_error32("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 Fjoin32(), this function can fail for lack of space (FNOSPACE) and can be re-issued again after allocating more space to complete the operation.

Fproj32

Fproj32() is used to update a buffer in place so that only the desired fields are kept (in other words, so that the result is a projection on specified fields).

```
int
Fproj32(FBFR32 *fbfr32, FLDID32 *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 "fld.tbl.h"
FBFR32 *fbfr32;
FLDID32 fieldid[20];
. . .
fieldid[0] = A;   /* field id for field A */
fieldid[1] = D;   /* field id for field D */
fieldid[2] = BADFLDID;   /* sentinel value */
if(Fproj32(fbfr32, fieldid) < 0)
      F_error32("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`).

Fprojcpy32

`Fprojcpy32`() is similar to `Fproj32`() but the projection is done into a destination buffer.

```
int
Fprojcpy32(FBFR32 *dest, FBFR32 *src, FLDID32 *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(Fprojcpy32(dest, src, fieldid) < 0)
       F_error32("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.

Fupdate32

`Fupdate32`() updates the destination buffer with the field values in the source buffer.

```
int
Fupdate32(FBFR32 *dest, FBFR32 *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 `Fjoin32`()). Fields in the destination buffer for which there are no corresponding fields on the source buffer are left untouched (as in `Fojoin32`()). Fields in the source buffer for which there are no corresponding field on the destination buffer are added to the destination buffer (as in `Fconcat32`()). For example:

```
if(Fupdate32(dest,src) < 0)
     F_error32("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.

Conversion Functions

FML32 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 `CFfindocc32`(), which, although it is a read operation, converts from the user-specified type to the native type before calling `Ffindocc32`().) The function names are the same as their non-conversion FML32 counterparts except that they have a "C" prefix.

CFadd32

The `CFadd32`() function adds a user supplied item to a buffer creating a new field occurrence within the buffer:

```
int
CFadd32(FBFR32 *fbfr32, FLDID32 fieldid, char *value, FLDLEN32 len,
int type)
```

where

- *`fbfr32`* 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(CFadd32(fbfr32,ZIP,"12345",(FLDLEN32)0,FLD_STRING) < 0)
      F_error32("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 `fbfr32.` (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(CFadd32(fbfr32,ZIP,&zipval,(FLDLEN32)0,FLD_LONG) < 0)
      F_error32("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`. `CFadd32`() returns `1` on success, and `-1` on error, in which case `Ferror32`() is set appropriately.

CFchg32

The function `CFchg32`() acts like `CFadd32`(), except that it changes the value of a field (after conversion of the supplied value):

```
int
CFchg32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc, char *value, FLDLEN32 len,
int type)
```

where

- *`fbfr32`* 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,

```
FLDOCC32 occurrence;
long zipval;
. . .
zipval = 12345;
occurrence = 0;
if(CFchg32(fbfr32,ZIP,occurrence,&zipval,(FLDLEN32)0,FLD_LONG) <
0)
       F_error32("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.

CFget32

`CFget32`() is the conversion analog of `Fget32`(). The difference is that it copies a converted value to the user-supplied buffer:

```
int
CFget32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc, char *buf, FLDLEN32 *len,
int type)
```

where

- *fbfr32* 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,

```
FLDLEN32 len;
. . .
len=sizeof(zipval);
if(CFget32(fbfr32,ZIP,occurrence,&zipval,&len,FLD_LONG) < 0)
        F_error32("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.

CFgetalloc32

`CFgetalloc32`() is like `Fgetalloc32`(); you are responsible for freeing the `malloc`’d space for the returned (converted) value with `free`:

```
char *
CFgetalloc32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc, int type, FLDLEN32
*extralen)
```

where

- *`fbfr32`* 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 `CFgetalloc32`() is shown as a character pointer data type (`char*` in C). The actual type of the pointer returned is the same as the type of the value to which it points.

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

```
char *value;
FLDLEN32 extra;
```

```
. . .
extra = 25;
if((value=CFgetalloc32(fbfr32,ZIP,0,FLD_LONG,&extra)) == NULL)
 F_error32("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.

CFfind32

`CFfind32`() returns a pointer to a converted value of the desired field:

```
char *
CFfind32(FBFR32 *fbfr32, FLDID32 fieldid, FLDOCC32 oc, FLDLEN32
len, int type)
```

where

- *`fbfr32`* 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 `CFfind32`() is shown as a character pointer data type (`char*` in C). The actual type of the pointer returned is the same as the type of the value to which it points.

Like `Ffind32`(), this pointer should be considered read only. For example:

```
char *CFfind32;
FLDLEN32 len;
long *value;
. . .
if((value=(long *)CFfind32(fbfr32,ZIP,occurrence,&len,FLD_LONG))== NULL)
   F_error32("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 `Ffind32`(), 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 `CFfind32`() 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 `Ffind32`(), which is guaranteed until the next modification of the buffer.

CFfindocc32

`CFfindocc32`() looks at occurrences of the specified field in 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).

```
FLDOCC32
CFfindocc32(FBFR32 *fbfr32, FLDID32 fieldid, char *value, FLDLEN32 len, int type)
```

where

- ♦ *fbfr32* 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 "fld.tbl.h"
FBFR32 *fbfr32;
FLDOCC32 oc;
char zipvalue[20];
. . .
strcpy(zipvalue,"123456");
if((oc=CFfindocc32(fbfr32,ZIP,zipvalue,0,FLD_STRING)) <  0)
        F_error32("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 `CFfindocc32`() converts the user-specified value to the native field type before examining the field values, regular expressions will work only when the user-specified type and the native field type are both `FLD_STRING`. Thus, `CFfindocc32`() has no utility with regular expressions.

Converting Strings

A set of functions (`Fadds32`(), `Fchgs32`(), `Fgets32`(), `Fgetsa32`(), and `Ffinds32`()) 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 `Ffinds32`() is the same as that described for `CFfind32`().

Ftypcvt32

The functions `CFadd32`(), `CFchg32`(), `CFget32`(), `CFgetalloc32`(), and `CFfind32`() use the function `Ftypcvt32`() to perform the appropriate data conversion. The synopsis of `Ftypcvt32`() usage is as follows (it does not follow the parameter order conventions):

```
char *
Ftypcvt32(FLDLEN32 *tolen, int totype, char *fromval, int fromtype, FLDLEN32
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`

Ftypcvt32() 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. Ftypcvt32() returns a pointer to the converted value, and sets *tolen to the converted length, upon success. Upon failure, Ftypcvt32() returns NULL. As an example of how Ftypcvt is used, the function CFchg32() is presented:

```
CFchg32(fbfr32,fieldid,oc,value,len,type)
FBFR32 *fbfr32;              /* fielded buffer */
FLDID32 fieldid;           /* field to be changed */
FLDOCC32 oc;               /* occurrence of field to be changed */
char *value;             /* location of new value */
FLDLEN32 len;              /* length of new value */
int type;                /* type of new value */
{
 char *convloc;          /* location of post-conversion value */
 FLDLEN32 convlen;         /* length of post-conversion value */
 extern char *Ftypcvt32;

         /* convert value to fielded buffer type */
  if((convloc = Ftypcvt32(&convlen,FLDTYPE(fieldid),value,type,len)) == NULL)
                return(-1);

  if(Fchg32(fbfr32,fieldid,oc,convloc,convlen) < 0)
                return(-1);
  return(1);
}
```

The user may call Ftypcvt32 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 (that is, when the values of both types are 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 defiines 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 `Finit32`() or `Falloc32`(), 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.

Fidxused32

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

```
long
Fidxused32(FBFR32 *fbfr32)
```

where *fbfr32* 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.

Findex32

The function `Findex32`() may be used at any time to index an unindexed fielded buffer:

```
int
Findex32(FBFR32 *fbfr32. FLDOCC32 intvl)
```

where

- *fbfr32* is a pointer to a fielded buffer
- *intvl* is the indexing interval

The second argument to `Findex32`() specifies the indexing interval for the buffer. If 0 is specified, the value `FSTDXINT` (defined in `fml32.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.

Frstrindex32

This function can be used instead of `Findex32`() in cases where the fielded buffer has not been altered since its index was removed:

```
int
Frstrindex32(FBFR32 *fbfr32, FLDOCC32 numidx)
```

where

- *`fbfr32`* is a pointer to a fielded buffer.
- *`numidx`* is the value returned by the `Funindex32` function.

Funindex32

`Funindex32`() discards the index of a fielded buffer and returns the number of index entries the buffer had before the index was stripped:

```
FLDOCC32
Funindex32(FBFR32 *fbfr32)
```

where *`fbfr32`* 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 = Funindex32(fbfr32);
   ```

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

   ```
   num_to_send = Fused32(fbfr32);
   ```

3. Send the buffer without the index:

   ```
   transmit(fbfr32,num_to_send);
   ```

4. Restore the index to the buffer:

   ```
   Frstrindex32(fbfr32,save);
   ```

On the receiving side, the index could be regenerated with the following statement:

```
Findex32(fbfr32);
```

Note that the receiving process cannot call `Frstrindex32`() 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 `Funindex32`(); this function either saves space when storing a buffer on a disk or reduces transmission costs 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.

Fread32 and Fwrite32

The I/O functions `Fread32`() and `Fwrite32`() work with the Standard I/O Library:

```
int Fread32(FBFR32 *fbfr32, FILE *iop)
int Fwrite32(FBFR32 *fbfr32, 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 `Fwrite32`(), like this:

```
if (Fwrite32(fbfr32, iop) < 0)
  F_error32("pgm_name");
```

A buffer written with `Fwrite32` may be read with `Fread32`(), as in:

```
if(Fread32(fbfr32, iop) < 0)
 F_error32("pgm_name");
```

Although the contents of the fielded buffer pointed to by `fbfr32` are replaced by the fielded buffer read in, the capacity of the fielded buffer (size of the buffer) remains unchanged.

`Fwrite32`() discards the buffer index, writing only as much of the fielded buffer as has been used (as returned by `Fused32`()).

`Fread32`() restores the index of a buffer by calling `Findex32`(). The buffer is indexed with the same indexing interval with which it was written by `Fwrite32`().

Fchksum32

A checksum may be calculated for verifying I/O:

```
long chk;
. . .
chk = Fchksum32(fbfr32);
```

The user is responsible for calling `Fchksum32`(), writing the checksum value out along with the fielded buffer, and checking it on input. `Fwrite32`() does not write the checksum automatically.

Fprint32 and Ffprint32

The function `Fprint32`() prints a fielded buffer on the standard output in ASCII format:

```
Fprint32(FBFR32 *fbfr32)
```

where `fbfr32` is a pointer to a fielded buffer

`Ffprint32()` is similar to `Fprint32()`, except the text is printed to a specified output stream:

```
Ffprint32(FBFR32 *fbfr32, FILE *iop)
```

where

- *`fbfr32`* 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. `Fname32()` 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.

Fextread32

`Fextread32()` may be used to construct a fielded buffer from its printed format, that is, from the output of `Fprint32()` (hexadecimal values output by `Fprint32()` are interpreted properly).

```
int
Fextread32(FBFR32 *fbfr32, FILE *iop)
```

`Fextread32()` accepts an optional flag preceding the field-name/field-identifier specification in the output of `Fprint32()`, 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

Table 5-4 Fextread Flags

flag	indicates
=	one field should be assigned to another
#	comment line - ignored

If no flag is given, the default action is to `Fadd32`() 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 `Ferror32`() is set accordingly. If end of file is reached before a blank line, `Ferror32`() is set to `FSYNTAX`.

Boolean Expressions of Fielded Buffers

The functions described in this section evaluate boolean expressions in which the “variables” are the values of fields in a fielded buffer. These functions allow you to:

- ♦ compile a boolean expression into a compact form suitable for evaluation
- ♦ evaluate a boolean expression against a fielded buffer, 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 each expression is evaluated. Table 5-5 shows the Backus-Naur Form (BNF) definitions of 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 Definitions of Boolean Expressions

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>

Table 5-5 BNF Definitions of Boolean Expressions

Expression	Definition
<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 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 `FLDTBLDIR32` and `FIELDTBLS32` environment variables are set, as described in Chapter 3, before using the boolean compilation function. The field types used in booleans are those allowed for FML32 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.

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 boolean compiler performs the following conversions:

- `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
- `constant`—a constant
- `(expression)`—an expression in parentheses

A field name or a field name followed by a subscript is a primary expression. The subscript indicates which occurrence of the field is being referenced. The subscript may be either an integer constant, or ? indicating any occurrence; the subscript cannot be an expression. If the field name is not subscripted, field occurrence 0 is assumed.

If a field name reference appears without an arithmetic, unary, equality, or relational operator, then its value is the long integer value 1 if the field exists and 0 if the field does not exist. This may be used to test the existence of a field in the fielded buffer regardless of field type (note that there is no * indirection operator).

A constant is a primary expression. Its type may be `long`, `double`, or `carray`, as discussed in the conversion section.

A parenthesized expression is a primary expression 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	< , >, <=, >=, ==, !=

Table 5-6 Boolean Expression Operators

Type	Operators
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 FML32 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 FML32 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 !=.

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. Unlike in C, however, it is *not* guaranteed that the second operand is not evaluated if the first operand is 0. 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. Consider the following example:

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

The expression is 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 is if `PETS` exists and any occurrence of it contains the characters “dog”.

Boolean Functions

The following sections describe the various functions that take boolean expressions as arguments.

Fboolco32

Fboolco32() compiles a boolean expression for FML32 and returns a pointer to an evaluation tree:

```
char *
Fboolco32(char *expression)
```

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

Space is allocated using malloc(3) to hold the evaluation tree. For example,

```
#include "<stdio.h>"
#include "fml32.h"
extern char *Fboolco32;
char *tree;
. . .
if((tree=Fboolco32("FIRSTNAME %% ’J.*n’ && SEX == ’M’")) == NULL)
  F_error32("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 Fboolco32() is used by the other boolean functions listed below; thus, the expressiondoes not have to be re-compiled constantly.

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.

Fboolpr32

Fboolpr32() 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
Fboolpr32(char *tree, FILE *iop)
```

where

- `*tree` is a pointer to a boolean tree previously compiled by `Fboolco32`
- `*iop` is a pointer of type `FILE` to an output file stream

This function is useful for debugging.

Executing `Fboolpr32`() on the expression compiled above would yield the following:

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

Fboolev32 and Ffloatev32

These functions evaluate a fielded buffer against a boolean expression.

```
int Fboolev32(FBFR32 *fbfr32,char *tree)

double Ffloatev32(FBFR32 *fbfr32,char *tree)
```

where

- *`fbfr32`* is the fielded buffer referenced by an evaluation tree produced by `Fboolco32`
- *`tree`* is a pointer to an evaluation tree that references the fielded buffer pointed to by `fbfr32`

`Fboolev32`() returns true (1) if the fielded buffer matches the boolean conditions specified in the evaluation tree. This function does not change either the fielded buffer or the evaluation tree. Using the evaluation tree compiled above, the following code would print "Buffer selected."

```
#include <stdio.h>
#include "fml32.h"
#include "fld.tbl.h"
FBFR32 *fbfr32;
. . .
Fchg32(fbfr32,FIRSTNAME,0,"John",0);
Fchg32(fbfr32,SEX,0,"M",0);
if(Fboolev32(fbfr32,tree) > 0)
  fprintf(stderr,"Buffer selected\n");
else
  fprintf(stderr,"Buffer not selected\n");
```

`Ffloatev32`() is similar to `Fboolev32`(), but returns the value of the expression as a double. For example, the following code would print "6.6."

```
#include <stdio.h>
#include "fml32.h"
FBFR32 *fbfr32;
. . .
main() {
  char *Fboolco32;
  char *tree;
  double Ffloatev32;
  if (tree=Fboolco32("3.3+3.3")) {
      printf("%lf",Ffloatev32(fbfr32,tree));
  }
}
```

If `Fboolev32`() were used in place of `Ffloatev32`() in the above example, a 1 would be printed.

CHAPTER

6 Examples

The BEA MessageQ kit includes an example of building, sending, receiving, and interpreting an FML32 message. Refer to `examples/x/x_fml.c` in your BEA MessageQ kit.

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