jEDI Programmer's Guide

Contents

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
BOLD	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates JBASE commands, keywords, and options; BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates JBase identifiers such as filenames, account names, schema names, and Windows NT filenames and pathnames.
UPPERCASE Italic	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, filenames, and pathnames.
`Courier`	Courier indicates examples of source code and system output.
`Courier Bold`	Courier Bold In examples, courier bold indicates characters that the user types or keys (for example, <Return>).
[]	Brackets enclose optional items. Do not type the brackets unless indicated.
{}	Braces enclose nonoptional items from which you must select at least one Do not type the braces.
ItemA | itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
. . .	Three periods indicate that more of the same type of item can optionally follow.
⇒	A right arrow between menu options indicates you should choose each option in sequence. For example, “Choose File ⇒Exit” means you should choose File from the menu bar, and then choose Exit from the File pull-down menu.

Syntax definitions and examples are indented for ease in reading.

All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.

Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

Introduction

This chapter describes how a third party software provider can write a new database driver, and interface it to existing jBASE applications. This is provided through the jEDI API (jBASE External Database Interface). It allows an application that performs its database requests through jEDI (for example, all jBASE applications used jEDI for their database access) to access databases not directly supported by jEDI itself.

NOTE: that non-jBASE applications can use jEDI as their database management system. This topic is described in the following chapter 'Calling jEDI from a non jBASE application'.

Supplied with jBASE are a number of standard database drivers built into the jEDI library code for jBASE data files and OS directories. It is possible to write alternate filing systems conforming to the jEDI API and all existing applications can then seamlessly use the new database through the new database drivers.

The directory $JBCRELEASEDIR/src contains examples of some of the functionality described in this document. The directory $JBCRELEASEDIR/include contains two source files to be included in user-written code, namely jsystem.h and jedi.h.

The source file jsystem.h provides all the necessary definitions for writing external 'C' functions and interfacing them with jBC source files. The source file jedi.h provides all the necessary definitions to allow users to write new database drivers.

The following code segment shows an example of a jBASE program accessing two files, and simply copying the records from one file to another:

```
InputFileName = "InputFile"
OutputFileName = "OutputFile"
OPEN InputFileName TO InputDSCB ELSE
   STOP 201,InputFileName
END
OPEN OutputFileName TO OutputDSCB ELSE
   STOP 201,OutputFileName
END
CLEARFILE OutputFileName
SELECT InputDSCB
LOOP WHILE READNEXT RecordKey DO
   CRT RecordKey
   READ Record FROM InputDSCB,RecordKey ELSE
      STOP 202,RecordKey
   END
   WRITE Record ON OutputDSCB,RecordKey
REPEAT
```

The above shows typical uses of database access statements available in jBC, such as OPEN, READ, and WRITE, CLEARFILE, SELECT and READNEXT. All of these statements cause, at program execution, a call to the jEDI API. The jEDI API will route the database request to the appropriate database driver for the file type. For example, the input file could be a j1 file while the output file could be a UNIX directory.

It is important to note that the application itself is unaware of the database type it is connected to. In some extreme circumstances it may specifically need to know, and can use the IOCTL() function to do so, but this use is rare.

The jEDI API has a number of database drivers built in. These include drivers for the various hashed databases , for treating UNIX directories like database files, optionally for C-ISAM and so on. For a basic jBASE installation, these drivers are sufficient for all your normal application needs, and no knowledge of jEDI, or setting up of jEDI is required.

It is possible to write your own database drivers than can be loosely bound to jEDI, such that the OPEN, READ, WRITE etc. calls from an application can be routed to your own code rather than to a supplied database drivers.

Shared Object Database Drivers

A shared object database driver is the mechanism by which a program using the jEDI API to access a file (such as a jBC program) can use a database driver, contained in a standard shared object, to provide the necessary database I/O functionality. The application that is calling the jEDI interfaces will remain unaware of where exactly the data is coming from, or from what sort of database the data is contained in.

There are six steps to creating a shared object database driver. An example of a shared object database driver is supplied with jBASE in the source file $JBCRELEASEDIR/src/jediDExample.c . It may also be useful to examine the supplied source $JBCRELEASEDIR/src/jediCExample.c , which shows an example of a ‘C’ program using calls to jEDI to perform database operations.

The following short example is a useful introduction to building a shared object database driver. It is recommended the reader attempt this example first of all.

Step 1. The first step involves creating an external database driver source. This of course is the hardest step and will be expanded upon in much more detail later on. For now though, we will use the supplied example source code. Use the following steps at shell:

```
% cd             ;# Change to home directory
% mkdir sosrc    ;# Create a source directory
% cd sosrc       ;# Change to the new source directory
% cp $JBCRELEASEDIR/src/jediDExample.c ./mydd.c
                     ;# Copy source to file ‘mydd.c’
```

If you look at the new source mydd.c , you will see that there is only one exported symbol, namely ExampleInit(). You can change this to be any name you like. For this short tutorial, we will assume the name is left at ExampleInit, as the name has significance later on.

Step 2. Compile the database driver and create a shared object out of it. The following steps at shell provide an example of this:

```
% cd $HOME/sosrc ;# Make sure in correct directory
% cc -c -I$JBCRELEASEDIR/include mydd.c
                ;# Compile source to an object
% jBuildSLib mydd.o -o $HOME/lib/mydd.so
                    ;# Create shared object out of mydd.o
```

Note that during the building of the shared object stage, the jBuildSLib command will create a shared object at $HOME/lib/mydd.so that contains a single object. You can include other objects on the command line so that mydd.so contains many objects. For example, you could execute:

```
% jBuildSLib mylib.a mydd.o newdd.o -o libfb.so
```

In the above example, the objects mydd.o and newdd.o, plus all the objects in the archive file mylib.a, will get built into a single shared object.

Step 3. Create a file definition. There needs to be a UNIX file that tells jBASE "This is really a reference to a file contained in a shared object database driver". You can create one using any UNIX mechanism, normally via an editor such as vi. However, the command at shell shown below is equally valid:

```
% echo "JBC__SOB ExampleInit .d" > $HOME/myfile
```

The above example creates a UNIX file at $HOME/myfile. The first 8 characters are always the same, and are "JBC__SOB" (Note there are two _ characters). The next part, `ExampleInit`, gives the name of the initialisation function in the database driver, as noted in Step (1). The remainder of the definition can be anything at all -- it is up to the database driver to interpret it how it likes. In the supplied database driver, we take the name of the UNIX file opened and append the remainder of the definition to arrive at the name of the file required. This mechanism will become clearer later on.

Step 4. Create the database itself. The example given simply uses flat UNIX files in a normal UNIX directory as a crude sort of database. Given the naming convention in step 3), we can create the database itself with the shell command:

```
% mkdir $HOME/myfile.d
```

Step 5. Create some data records. We will create 3 data records of zero length with the following simple UNIX shell commands:

```
% touch $HOME/myfile.d/reca
% touch $HOME/myfile.d/recb
   % touch $HOME/myfile.d/recc
```

Step 6. Either create your own jBC program to access the database in $HOME/myfile.d, or use an existing application. For example,

```
% LIST myfile
PAGE    1                        11:46:55  30 AUG 1994

myfile........

reca
recb
recc

     3 Records Listed
```

Some assumptions have been made in this example:
That the directory $HOME is a component part of the JEDIFILEPATH environment variable, or the JEDIFILEPATH variable is not set.
That the directory $HOME/lib is a component part of the JBCOBJECTLIST environment variable or the JBCOBJECTLIST variable is not set.
At this stage it is worthwhile understanding the chain of events, which occur when accessing the newly created shared object database driver in the above example. The mechanism inside the jBASE run-time library is as follows:

- The OPEN “myfile” statement in the jBC source generates a function call to the JLibFOPEN function, as supplied by the jBASE run-time libraries.
- The JLibFOPEN function generates a call to the generic OPEN function inside the jEDI library.
- The jEDI OPEN function will attempt to open a UNIX file called “myfile” in all the directories that are contained in the JEDIFILEPATH environment variable. Eventually the UNIX file $HOME/myfile will be opened.

- The jEDI OPEN function will now ask all the established database drivers if the file is one that belongs to their type. The database driver for shared objects will understand this, as the first 8 characters are JBC__SOB.

- The shared object database driver will attempt to use code in the jBC run-time library to execute a function taken from shared objects called “ExampleInit”. The mechanism happens to be the same mechanism that is used for the CALL statement. This will succeed, and the ExampleInit() function will establish itself as a database driver in its own right.

- The shared object database driver will now pass control to the OPEN function defined when the database driver was established in the call to ExampleInit. Control of the OPEN now passes outside of jEDI and into the user-written OPEN code.

- The user-written OPEN code now has freedom to do anything it likes. In the example code, it will simply concatenate the strings “$HOME/myfile” and “.d” to create a name “$HOME/myfile.d” and expect this to be an existing UNIX directory.

- Assuming the user-written OPEN returned a successful code, then the OPEN statement now completes. Any future database operations such as READ, WRITE and so on will now be passed to the functions defined by the ExampleInit() function as being responsible for the various operations.

-

The remainder of this topic now details all the functions required to create a shared object database driver confirming to the jEDI API. They would normally be in one source code. Only a single function is required to be visible, the remainder can be defined as ‘static’.

The functions can be any name whatsoever, and any names described in this document are purely examples. The only consideration is that the visible initialisation function name is referenced in the file definition. For example, if you changed the function name ExampleInit to MyInit, then the file definition described in Step 3 previously would have to reference MyInit instead of ExampleInit.

The handle for a jEDI opened file is of type JediFileDescriptor and is defined in the supplied source file $JBCRELEASEDIR/include/jedi.h. Once the file is opened, this handle is passed to the jEDI API and so to the database driver for all future database operations on the file.

The jEDI interface does not provide opportunities to create the file nor to delete the file. It is assumed that the database to which you are interfacing has its own facilities to do this.

The function required to build a database driver conforming to the jEDI API are:

1. Database driver initialisation. This is the only visible function in the database driver source code, the remainder will probably be declared as static functions (not visible outside the source code). It is called when the first file is opened in the program, and is responsible for establishing itself as a database driver.
2. Open file. Performs all necessary functionality to open a file.
3. Close file. Called when a file is closed.

4. Select record keys in a file. The application wants to select some or all of the records in the file.
5. Terminate the selection process. The selection process is terminated.
6. Read the next record from a selection. Following the selection call, the application will repeatedly call this to read the next record key selected.
7. Read a record from the file. Read a single record from the database.
8. Write a record to the file. Write a single record to the database.
9. Delete a record from the file. Delete a single record from the file.
10. Clear all records from the file. Delete all records from the file.
11. Provide locking support. Provide the record locking mechanism to support record locks defined in the application.
12. Provide database specific IOCTL functionality. General functionality mostly unique to the database driver itself, in the same manner the ioctl() function is fairly device dependent in the 'C' library code.
13. Synchronize the database (i.e. flush any cache data). This function is called when the application wants to 'checkpoint' the database so forcing any data in the cache buffers to disk.

File ajar processing

There is a limit to the number of files that a process can have open at any one time. Depending upon the operating system and its configuration, this is often in the region of 50 to 200. Unfortunately, this clashes with many applications that require hundreds of files open simultaneously.

The mechanism to work around this call called file ajar processing, and involves temporarily closing the file to the operating system, but the application still believes the file is opened. Then, when a database operation is requested on a file that is temporary closed (or ajar), then a different file is closed and made ajar, and the requested file is re-opened. This mechanism works on a least used file algorithm so that the most frequently used files do not suffer the performance penalty.

This ability to temporarily close and reopen files depends upon the database driver itself, and a database driver can choose to implement it or not.

The jEDI API provides this functionality itself, but uses two extra functions supplied by the database driver to accommodate this. During the OPEN function call, detailed later, there are two members to the file descriptor called ProcessAjar and ProcessReopen. By default these members are initialised to NULL and the database driver will not support ajar file processing. If you want to support file ajar processing, then these two members need to be set to the address of two support functions within the database driver to handle the following functionality.

File descriptor member ProcessAjar.

Set the member ProcessAjar in the file descriptor (see the OPEN function description later) to the address of a database driver support function that will be called by the jEDI API each time a file is required to be temporarily closed. The description of this function supplied by the database driver is :
Synopsis:

```
static  void MakeFileAjar( FileDescriptor )
JediFileDescriptor     * FileDescriptor ;
```

PARAMETERS:

FileDescriptor (Input and Output parameter) this is the file descriptor that was returned when the file was originally opened.

RETURN VALUE:

None.

OPERATION:

The address of this function is made available to jEDI during the OPEN function described later. For example, in the OPEN function you might do:

```
FileDescriptor->ProcessAjar = MakeFileAjar ;
```

Hence, the actual name of the function is not important, as jEDI will do the call through a function pointer.
This function should perform all necessary functionality to temporarily close the file and return. It is assumed the function always succeeds. If it fails, no error is reported and jEDI will try to make ajar another file.
The jEDI code will, by default, assume a UNIX file at FileDescriptor->PathName with a UNIX file descriptor at FileDescriptor->FileFd is part of the file operation, and will close these automatically. Remember that the database driver should only support this type of functionality if closing the files doesn't cause any harm, such as losing any UNIX file locks that were set.

File descriptor member ProcessReopen.

Set the member ProcessReopen in the file descriptor (see the OPEN function description later) to the address of a database driver support function that will be called by the jEDI API each time a file is that has previously been temporarily closed needs to be re-opened. The description of this function is:

Synopsis:

```
static  int ReopenAjarFile( FileDescriptor )
    JediFileDescriptor     * FileDescriptor ;
```

PARAMETERS:

FileDescriptor. (Input and Output parameter) This is the file descriptor that was returned when the file was originally opened.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

OPERATION:

The address of this function is made available to jEDI during the OPEN function described later. For example, in the OPEN function you might do:

```
FileDescriptor->ProcessReopen = ReopenAjarFile ;
```

Hence, the actual name of the function is not important, as jEDI will do the call through a function pointer.
This function should re-open the file previously made ajar.
The jEDI code will, by default, assume a UNIX file at FileDescriptor->PathName with a UNIX file descriptor at FileDescriptor->FileFd is part of the file operation, and will re-open these automatically.

jEDI support functions

This section describes some of the support functions available to the database driver. In particular, note the mandatory use of JediMalloc() instead of the normal malloc() function, and similar memory allocation routines.

Memory allocation functions

When an application connects to jEDI using function JediConnect (see the chapter "jEDI API calls") it provides a list of memory allocation routines for use by the database driver. This will normally be the standard function such as malloc() , free() and so on, but there are specialized circumstances when this will not be the case.
Therefore a database driver should not use the memory allocation functions such as malloc(), realloc() etc.. The driver should instead use a supplied alternative set, where the first parameter passed is the file descriptor variable that was passed to the database driver. The calls to use are :

```
void * JediMalloc(JediFileDescriptor * fd ,
              int size);
void * JediRealloc(JediFileDescriptor * fd ,
              void * source , int newsize );
void * JediCalloc(JediFileDescriptor * fd ,
              int qty , int size );
void * JediStrdup(JediFileDescriptor * fd ,
              void * source ) ;
void   JediFree(JediFileDescriptor * fd ,
              void * source ) ;
void * JediReadMalloc(JediFileDesciptor * fd ,
              int size);
```

The functions JediMalloc, JediRealloc, JediCalloc, JediStrdup and JediFree are the replacement functions to be used by the database driver instead of malloc, realloc, calloc, and strdup and free respectively.
The function JediReadMalloc is to be used by the READ function in the database driver when it is re-sizing a buffer that was originally supplied. The use of JediReadMalloc is detailed later in the READ function description.

Record locking function

The database driver has to supply a LOCK function so that record-locking functionality can be provided. The database driver can choose to do whatever it likes with these requests. However, the database driver may wish to make calls to the same locking function as used by jBASE internal database drivers.

```
int JediSystemLock(JediFileDescriptor * fd ,
        int command , int hash ) ;
```

fd: The file descriptor that was passed to the LOCK function. The following members of the file descriptor must be initialised (usually done by the database driver in the OPEN code) :

- **FileId1 and FileId2**. Two integers that uniquely describe the file. See the description of the OPEN function.
- **FileFd**. The UNIX file descriptor of the opened file. If the jRLA demon is active, then this entry is ignored. Otherwise, we use UNIX locks against the file descriptor in this entry.
- **command**: One of the following to describe the action to take :

- **JEDI_LOCK**: Take a record lock. If the lock is already taken by another process, then the calling process will wait until it can gain control of the lock.
- **JEDI_LOCK_NOWAIT**: Take a record lock, but if the lock is already taken by another process, then return immediately.
- **JEDI_UNLOCK**: Release a record locks. No operation is performed if the lock doesn't already exist, or it does exist but is taken by another process.
- **JEDI_UNLOCK_ALL**: Release all the locks taken for this file. The 'hash' parameter is ignored.
-

hash: This is a 32-bit integer that describes the record key to lock or unlock. It is ignored if the command value is JEDI_UNLOCK_ALL. This means it doesn't support full record locking, but a simpler scheme with only an extremely remote chance that hash values will clash for different record keys. The hash value can be anything the database driver likes. The database drive can use the JediBaseHash function (see later) to convert a record key into a 32-bit hash value.
The function can return the following values :

- **0** : Shows the operation was successful.
- **JEDI_ERRNO_LOCK_TAKEN** : The command was JEDI_LOCK_NOWAIT and the requested lock was already taken by another process.
- **Any other value** : This is a fatal error. The values are usual UNIX error numbers, described in header file errno.h .

jEDI base code

There is some functionality provided by jEDI for all database drivers and this is provided within jEDI without the need for the database drivers to provide it themselves :

- Database journaling. Updates to the database, such as record updates , file clears and so on will be journaled automatically by jEDI assuming the facility is operational. The database driver need not do any specific operation. There are options in the code for the OPEN to prevent a particular file from being journaled if required.
- Transaction boundaries. Within jEDI the notion of single level transaction boundaries are supported. This is done by the base code of jEDI. The database driver need not do any specific operation. There are options in the code for the OPEN to prevent a particular file from being part of a transaction if required.
- File ajar processing. The routine closing and re-opening of files on a temporary measure is done by the jEDI API , and was described fully in the earlier section 'File ajar processing' .

INIT: Initialisation of database driver.

The first function to write is the main initialisation function and will be called just once, the first time the database driver is referenced. It should set up a structure that defines the database driver, and then

call a jEDI function to register the database driver. The name of this function, as explained earlier, can be anything but the name you decide is the name to specify in the UNIX file that is the file descriptor. For example, if you name this function INIT, then typically the UNIX file that describes the shared object database driver would be :

```
% COUNT ./THISFILE

169 RECORDS COUNTED
% cat ./THISFILE
JBC__OBJ INIT XYZ-INFO^TBAGS
```

SYNOPSIS:

```
int  INIT()
```

PARAMETERS:

None.

RETURN VALUE:

The return value is the database driver number this has been registered as (using function call JediBaseAddDriver()).
If an error occurs, then return a negative value and set errno set to show the reason for the failure.

OPERATION:

The code for this function will look pretty much the same for each database driver. While the user can of course add any extra initializations they like, the bulk of the code will look like this:

```
int  OPEN() , CLOSE() , SELECT() , SELECTEND() ;
int  READNEXT() , READ(), WRITE();
int  DELETE() , LOCK() , IOCTL() ;
int  CLEAR() , SYNC() ;
struct  JediDriverStruct  p1 ;
int  DriverNumber ;
/*
 * Initialise the members of the 'p1' structure
 * with our device driver function addresses.
 */
memset(&p1,0,sizeof(p1));
```

```
p1.FileTypeDescr = "OurDatabase" ;
p1.OpenCodePtr = OPEN ;
p1.JediClose = CLOSE ;
p1.JediSelect = SELECT ;
p1.JediSelectEnd = SELECTEND ;
p1.JediReadnext = READNEXT ;
p1.JediReadRecord = READ ;
p1.JediWriteRecord = WRITE ;
p1.JediDeleteRecord = DELETE ;
p1.JediLock = LOCK ;
p1.JediIOCTL = IOCTL ;
p1.JediClearFile = CLEAR ;
p1.JediSync = SYNC ;
/*
 * Add this to the list of supported
 * database drivers and return.
 */
return JediBaseAddDriver(&p1);
```

Basically, this function sets up a structure with details of all its functions to support, all the required database I/O requests, and then call JediBaseAddDriver() to register it as a database driver.
Once this initialisation code has completed successfully, then any database I/O requests for this database driver will be re-directed to the functions declared in the above code.

OPEN: Open a File.

This function is called whenever the application performs an OPEN request for this particular database driver.
Synopsis:

```
static  int  OPEN(FileDescriptor , HeaderInfo ,
                    HeaderInfoLen )
JediFileDescriptor  * FileDescriptor;
char                * HeaderInfo ;
int                   HeaderInfoLen ;
```

PARAMETERS:

FileDescriptor. (Input and Output parameter) This is the address of a file descriptor that has been partially filled in by the calling jEDI code. The members of the structure filled in before the call to OPEN, and the expected members to be filled in if the OPEN succeeds, are detailed later.
HeaderInfo. (Input parameter) This is a pointer to the first 'n' bytes of the UNIX file that was opened. For example, in the case of shared object database drivers, then this would typically point to

```
JBC__SOB INIT AnyOtherCode
Second line of description
```

HeaderInfoLen. (Input parameter) This describes the amount of data pointed to by HeaderInfo.

RETURN VALUE:

0 shows the file was opened successfully.
ENOENT shows the file could not be opened but there is no reason why other database drivers should not attempt to open the file.
Any other positive value if a fatal error occurred and no more database drivers will be given the opportunity to open the file.

OPERATION:

The OPEN function will now use the information passed in the above three parameters to try to open the relevant file in the foreign database it is accessing. If the OPEN succeeds, then the database driver will fill in other parts of FileDescriptor and return a value of 0.
How you make the correspondence between the information in FileDescriptor and the foreign database file name is entirely up to the driver. For example, the UNIX file may look like this:

```
JBC__SOB INIT -rcoffice -ausers CUSTOMERS
```

The database driver may interpret this to mean to open the file CUSTOMERS in remote machine coffice in account users. The example program provided with jBASE, as described earlier, simply concatenates the actual UNIX file name opened with the operands that follow the database function name INIT.
Any security and access permissions is entirely the responsibility of the database driver. Just because the user has had access to the file definition doesn't necessarily mean the user is entitled to access the file. It would be too easy for a user simply to edit their definition of the file and then try to open it.

NOTES:

As mentioned earlier, the information for the user is mostly passed in parameter ‘FileDescriptor’. The following is a list of members of FileDescriptor that are initialised by the jEDI code before calling INIT:

- ProcessId. The process id of the calling process.
- CurrentDir. The current working directory of the process.
- PathName. The path name of the file description originally opened.
- OptionalArgs. The list of text that followed the ‘JBC__SOB FuncName’ text in the file description originally opened.
- StatInfo. The details returned by function stat() describing the file description originally opened.
- Status. Set to one or more of the JEDI_STATUS_READ and JEDI_STATUS_WRITE bits to show if the original file description was opened in read only mode, or read+write mode.
- The following is a list of members of FileDescriptor that INIT should fill in , assuming the open is successful:
- ConvertFlags. Some database drivers will perform conversions on the data read in. For example, when reading records from a UNIX directory, the new-line characters will be replaced with attribute marks (i.e. 0x0a characters replaced with 0xfe). If the database driver supports conversion of data, it should initialise this entry with one or more of the JEDI_CONVERT_xxx bits defined in jedi.h . These bits can subsequently be modified by the application by a call to IOCTL.
- FilesUsed. Enter here the approximate number of system files that will remain open for the file. This allows the ajar processing to try to estimate when files need closing. The value here need not be accurate, it will just be a minor performance hit if wrong.
- Status. Set additional bits to this field. Note that some may already be initialised by jEDI, so the following bits should be OR’ed :

  i. JEDI_STATUS_FIELDREAD Set if the READ function can perform field read operations.

  ii. JEDI_STATUS_FIELDWRITE Set if the WRITE function can perform field write operations .

  iii. JEDI_STATUS_CANBEAJAR Set if this file supports ajar processing.

  iv. JEDI_STATUS_USING_RLA Set if the file driver is using the jBASE supplied JediSystemLock locking mechanism. See earlier description of the JediSystemLock() function call.

v. JEDI_STATUS_WRAPUP_LOCKS_ONLY When a process terminates, then if this bit is set the jBC run-time code will not call the CLOSE function for this file, but will merely ensure that the locks for this file are released.

vi. JEDI_STATUS_READ User is allowed to READ from the file.

vii. JEDI_STATUS_WRITE User is allowed to WRITE to the file.

-

- TypePtr. This is a 'void *' pointer and allows the database driver to optionally insert any address here of their choosing. This is typically used to store a control block for the file unique to the database driver.
- FileFlags. Controls operation of the file, and can be one or more of the following bits

-

viii. JEDI_FILE_NOLOG If set, then updates to the file will not be sent to the jEDI database journaler.

ix. JEDI_FILE_NOTRANS If set, then updates to the file will not become part of any transaction.

x.

- FileType. Any unique integer here, for the use of the database driver only, so long as it doesn't clash with any internally defined values (they have a value less than 256).
- LockId1. If using the JediSyslockLock() function, then this is the first of two integers required to uniquely identify the file. This is usually set to the expression 'FileDescriptor->StatInfo.st_dev & 0xffff'.
- LockId2. If using the JediSystemLock() function, then this is the second of two integers required to uniquely identify the file. This is usually set to the expression 'FileDescriptor->StatInfo.st_ino' .
- ProcessAjar. If the database driver supports ajar processing, then this is the address of the function to call to change the status of a file from 'open' to 'ajar'. This mechanism was described in the earlier section 'File Ajar Processing'.
- ProcessReopen. If the database driver supports ajar processing, then this is the address of the function to call to change the status of a file from 'ajar to 'open'. This mechanism was described in the earlier section 'File Ajar Processing'.

CLOSE: Close an Opened File.

This function is called when a file descriptor is closed.

SYNOPSIS:

```
static  int  CLOSE(FileDescriptor , Flags)
JediFileDescriptor  * FileDescriptor;
   int                  Flags ;
```

PARAMETERS:

FileDescriptor. (Input and Output parameter) This is the file descriptor that was returned when the file was originally opened.
Flags. (Input parameter) This parameter is not used at present.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

OPERATION:

The close operation should typically release all resources that were allocated uniquely by the database driver itself. This includes closing down file descriptors, flushing any cache data, releasing workspace uniquely allocated to the file descriptor that was allocated by the database driver etc.
The file descriptor may be associated with SELECT structures allocated against the file descriptor. Therefore, any SELECT structures that were allocated against the file descriptor (see “SELECT” later) must be terminated. Note there can be more than one SELECT associated with a single file descriptor, and you should traverse the linked list cleaning up these multiple selects. An example of this is given in the source file jediDExample.c.

Once the close functionality has cleaned up its own allocated resources, it must call function JediFreeFileDescriptor as shown below:

```
JediFreeFileDescriptor(FileDescriptor);
```

This will release all the resources allocated by jEDI for the file descriptor, including the structure pointed to by FileDescriptor. Therefore, this call is the last operation you should perform.
NOTES:
The database driver should remove all locks associated with this file before actually closing a file. This is often achieved with a call to the LOCK function.

SELECT: Select Record Keys from a File.

This function is called when the user wants to perform some sort of selection of the file. At present, the select function is called from a jBC program to select all records in the file. However, calls from 'C' programs and future jBC programs may add selection criteria.

SYNOPSIS:

```
static  int SELECT(FileDescriptor , SelectPtr ,
                    SelectDetails, Index )
JediFileDescriptor    *  FileDescriptor;
struct JediSelectPtr  ** SelectPtr;
char                  *  SelectDetails;
int                      Index ;
```

PARAMETERS:

FileDescriptor. (Input and Output parameter) This is the file descriptor that was returned when the file was originally opened.
SelectPtr. (Input and Output parameter) The caller passes here the address of a "struct JediSelectPtr *" variable. If the SELECT works, then this function will return here the address of an allocated JediSelectPtr structure. This address will be passed to future calls to SELECTEND and READNEXT.
SelectDetails. (Input parameter) This is a NULL terminated string describing the selection details. For example, to select all record keys that match the regular expression "^MJI.*PIPE" the caller will set this variable to point to the string "^MJI.*PIPE". Note that none of the supplied database drivers as of release 1.9 support this feature. Future database driver may do so. Currently the language for jBC does not support this either, but again may do so in the future.
Index. (Input parameter) For database drivers that support multiple indexed files, this shows the index number to perform the select against. As of release 1.9, the jBC language does not directly support multiple indexed databases. However, this may change in future releases.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h .

OPERATION:

The SELECT function will allocate a structure of its own design that describes this selection call. The application will then make repeated calls to the READNEXT function to retrieve the record keys as designated by this SELECT process call. It is up to the designer of the database driver to decide if to make a list of all the record keys during this function call, or just to establish selection criteria and let the READNEXT do the actual selection.

A segment of the code to achieve the SELECT would look like this:

```
struct JediSelectPtr  *BlockPtr , **LoopPtr;
/*
 * Allocate space for the standard
 * JediSelectPtr structure.
 */
if ((BlockPtr=JediMalloc(
        FileDescriptor,
        sizeof(struct JediSelectPtr)
        )) == NULL)
{
    return errno ;
}
/*
 * Now to allocate a structure for our own
 * use. The structure name is
 * called Select, and it can be anything our
 * own database driver wants
 * it to be. The 'TypePtr' member of the
 * previously allocated structure will
 * contain a pointer to it.
 */
if ((BlockPtr->TypePtr=JediMalloc(
         FileDescriptor,
         sizeof(struct Select)
         )) == NULL)
{
    return errno ;
}
/*
 * The new structure 'BlockPtr' now needs to
 * be added to the end of
 * a linked list of selection structures
 * allocated against
 * this file descriptor.
 */
LoopPtr = &FileDescriptor->SelectPtr;
while(*LoopPtr != NULL)
{
    LoopPtr = (struct JediSelectPtr **)(*LoopPtr);
}
```

```
*LoopPtr = BlockPtr ;
/*
 * Set up our 'NextPtr' to be NULL to show we
 * are at end of linked list.
 */
BlockPtr->NextPtr = NULL;
/*
 * Return the address of the select
 * structure allocated to the caller.
 */
    *SelectPtr = BlockPtr ;
    return 0;
```

In the above code, the first memory allocation (using JediMalloc()) is the standard structure that needs to be allocated. This structure is the one passed to the SelectEnd and Readnext functions (described below).

The second memory allocation is optional and is a structure entirely of the making of the database driver. This allows the database driver to have its own data associated with a selection.

Note that there can be multiple selections associated with a single file descriptor. For example, a jBC source program could contain the following:

```
OPEN "FileName" TO FileVar ELSE STOP 201,"FileName"
SELECT FileVar TO SelectList1
.....
SELECT FileVar TO SelectList2
```

In the above example there will have been two calls to the SELECT function, both calls passing the same file descriptor. Therefore, on the linked list pointed to by `FileDescriptor->SelectPtr` there will be two select structures. Your SELECT, SELECTEND and CLOSE functions must be able to handle these instances.

SELECTEND: Terminate a Selection Process.

This function is called when the selection process is terminated. For example, in a jBASIC program this could be in one of the two ways:

```
OPEN “FileName” TO FileVar ELSE ...
SELECT FileVar TO SelectVar
LOOP WHILE READNEXT RecordKey FROM SelectVar DO ......
REPEAT
Or
OPEN “FileName” TO FileVar ELSE ...
SELECT FileVar TO SelectVar
......
SelectVar = 0
```

In the first example the selection process is exhausted. In the second example, the selection process is terminated prematurely,

SYNOPSIS:

```
static  int SELECTEND(FileDescriptor , SelectPtr )
JediFileDescriptor          * FileDescriptor;
    struct JediSelectPtr        * SelectPtr;
```

PARAMETERS:

FileDescriptor. (Input and Output parameter) This is the file descriptor that was returned when the file was originally opened.
SelectPtr. (Input and Output parameter) This is the address of the structure that was created during the SELECT function.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

OPERATION:

The select has come to an end, and this function should clear up all the space allocated during the SELECT function call, and remove the SelectPtr structure from the linked list of select structure pointed to somewhere in the FileDescriptor structure.

In the example code with the call to function SELECT(), there were two structures allocated, and the first was positioned at the end of the linked list. Carrying on from that example, the following code would clean up the resources allocated during the call to SELECT().

```
struct  JediSelectPtr  *Next , ** LoopPtr ;
/*
 * Free the structure allocated specifically
 * for the use of this database driver.
 * This can be used for whatever purposes
 * the database driver likes.
 * It is at this stage the database driver
 * would clean up all the members
 * pointed to by SelectPtr->TypePtr.
 */
JediFree(FileDescriptor,SelectPtr->TypePtr);
/*
 * Extract the pointer to the next structure
 * in the linked list.
 */
Next = SelectPtr->NextPtr ;
/*
 * Now to free the other structure allocated in
 * the call to SELECT()
 */
JediFree(FileDescriptor,SelectPtr);
/*
 * The structure passed to us is part of a
 * linked list of workspaces that
 * begin at address FileDescriptor->SelectPtr.
 * The address of our workspace is given
 * by the variable 'SelectPtr' and the forward
 * pointer for our linked workspace is
 * given by the variable 'Next'.
 *
 * We now have to go through the list and remove
 * our workspace from the linked list.
 */
LoopPtr = &FileDescriptor->SelectPtr;
for(;;)
{
    if (*LoopPtr == NULL) break;
```

```
    if (*LoopPtr == SelectPtr)
    {
        *LoopPtr = Next ;
        break;
    }
    LoopPtr = (struct JediSelectPtr **)(*LoopPtr);
}
/*
 * Finished, so return a good completion code.
 */
return 0 ;
```

READNEXT: Get Next Record Key from a Selection.

This function is typically called many times following a call to the SELECT function. This function will return the next record that has been selected following a call to the SELECT function.

SYNOPSIS:

```
static  int READNEXT(FileDescriptor , SelectPtr ,
              RecordKeyPtr , RecordKeyLenPtr)
JediFileDescriptor      *  FileDescriptor;
struct JediSelectPtr    *  SelectPtr;
char                    ** RecordKeyPtr ;
int                     *  RecordKeyLenPtr ;
```

PARAMETERS:

FileDescriptor. (Input parameter) This is the file descriptor that was returned when the file was originally opened.

SelectPtr. (Input parameter) This is the address of the structure that was created during the SELECT function.

RecordKeyPtr. (Input and Output parameter) The application passes in the parameter a pointer to a pointer to a buffer. This is where the next selected record key will be returned. If the buffer is not large enough (See RecordKeyLenPtr), then the READNEXT function should allocate sufficient memory and return at RecordKeyPtr the address of the allocated memory. It is up to the calling application to detect if this has been performed, and to free any allocated memory.

RecordKeyLenPtr. (Input and Output parameter) This is the address of an integer that describes the length of the buffer given by the expression "*RecordKeyPtr". When we have selected the next record key and placed it in the buffer "*RecordKeyPtr", we return here the length of the record key. If there are no more record keys, then -1 is returned here.

RETURN VALUE:

0 if the operation succeeded. This includes when the list of record keys are exhausted (in this case, *RecordKeyLenPtr set to -1).

Any other value shows the reason the operation failed using the values defined in errno.h.

OPERATION:

This function simply reads the next record key from the select list and returns details of the record key. At the end of processing all keys, we return -1 in the RecordKeyLenPtr variable.

This process is best illustrated by example. The example below will simply return 3 record keys called RecordKey1, RecordKey2 and RecordKey3. It is assumed that the previous call to SELECT will have resulted in the member's maxkey being set to 3 and currkey set to 0.

```
struct  Sel {
   int   maxkey ;
   int   currkey ;
   } ;

struct  Sel *OurPtr ;
char    recordkey[128];
int     recordkeylen ;
/*
 * Extract the structure for SELECTs that is unique
 * to our database driver.
 */
OurPtr = ((struct Sel *)SelectPtr->TypePtr);
/*
 * We simply return up to 3 record keys, namely
 * "RecordKey1" , "RecordKey2" and "RecordKey3".
 * Check to make sure that we have not already
 * returned 3 keys.
 */
if (++OurPtr->currkey > OurPtr->maxkey) {
   /*
    * No more valid record keys, so return -1
    * in the length field.
    */
   *RecordKeyLenPtr = -1;
   return 0 ;
}
/*
 * Create our actual record key in an
 * automatic variable
 */
recordkeylen = sprintf(recordkey,
                "RecordKey%d",OurPtr->currkey);
/*
 * Make sure the length of the record key
 * buffer is big enough.
 */
if (recordkeylen > *RecordKeyLenPtr)
{
   if ((*RecordKeyPtr = JediMalloc(
         FileDescriptor,recordkeylen)) == NULL)
```

```
   {
      return errno;
   }
}
/*
 * Return the actual record key and length
 * of record key.
 */
memcpy(*RecordKeyPtr,recordkey,recordkeylen);
*RecordKeyLenPtr = recordkeylen ;
return 0 ;
```

READ: Read a Record from a File.

This function is called when the application wants to read a record from the opened file.

SYNOPSIS:

```
static  int READ(FileDescriptor , Flags ,RecordKey,
                 RecordKeyLen , BufferPtr,
                 BufferLenPtr , FieldNumber)
JediFileDescriptor    *  FileDescriptor;
int                      Flags ;
char                  *  RecordKey ;
int                      RecordKeyLen ;
char                  ** BufferPtr ;
int                   *  BufferLenPtr ;
int                      FieldNumber ;
```

PARAMETERS:

FileDescriptor. (Input parameter) This is the file descriptor that was returned when the file was originally opened.
Flags. (Input parameter) One or more of the following bits

JEDI_RECORD_FIELDREADWRITE. If set, then the value at parameter ‘FieldNumber’ contains the field number (or attribute) to read, instead of the entire record. If the database drive doesn’t support the reading of individual fields, then it should have not set the JEDI_STATUS_FIELDREAD bit during the OPEN function (see the OPEN function for details of the ‘Status’ member of JediFileDescriptor) and this can be ignored.

RecordKey. (Input parameter) Pointer to a character array describing the record key to read in the array does not need to be 0 terminated.
RecordKeyLen. (Input parameter) The length of the RecordKey parameter
BufferPtr. (Input and Output parameter) This is the address of a character array where we will place the record data. The length of this is given by the BufferLenPtr parameter. Should this buffer not be large enough to accommodate the record, we will allocate more space using JediReadMalloc() and return here the address of the data space allocated. Thus the calling function can determine where the record was placed, and thus whether it was placed in the character array supplied to READ, or the character array allocated by READ. It is up to the calling function to free any allocated space using JediFree() if necessary.
BufferLenPtr. (Input and Output parameter) This is the address of an integer, originally set up to describe the size of the character array pointed to by ‘* BufferPtr’ . Upon return from READ, this will indicate the size of the record that was read in.

FieldNumber. (Input parameter) If the caller wishes to read an individual field instead of the entire record, then they will set the JEDI_RECORD_FIELDREADWRITE bit in the Flags parameter and set this parameter to be the field number to read in. If the database driver does not support this operation, you can ignore this field (see earlier description of JEDI_RECORD_FIELDREADWRITE).

RETURN VALUE:

0 is returned if the record is read in successfully.
ENOENT is returned if the record does not exist.
Any other value shows the reason the operation failed using the values defined in errno.h .

> Note: Although the database driver is only responsible for the above return values, the jEDI code may return other codes to the calling application outside the control of the database driver. These codes are EDEADLK (when a deadly embrace has been detected and avoided), and JEDI_ERRNO_LOCK_TAKEN (when a lock with NO WAIT was specified, and the lock was already taken by another process).

OPERATION:

The READ function should, by default, read in an entire record. If the bit defined by (Flags & JEDI_RECORD_FIELDREADWRITE) is set, then the database driver should just return a single field from the record, as given by parameter FieldNumber. Note that if database driver doesn't want to support reading of individual fields, then it should not set the JEDI_STATUS_FIELDREAD bit in the Status field of the file descriptor during the OPEN function call, leaving it up to the application to extract the individual field from the entire record.
It will return the data at the address *BufferPtr and return the length of the record at *BufferLenPtr. The database accessed by the database driver will probably have an entirely different record structure to that expected by the application. Therefore it is up to the database driver to convert the record from the foreign database into a format understandable by the calling application. For example, if the database had defined the record layout as:

```
INT      UserId    4 byte integer field
STRING   Name(20)  20 character name field.
FLOAT    Balance   8 byte monetary value field.
```

Then it would be up to the database driver to convert it to a format acceptable to the calling application. Using the above example, the following code would perform the conversion:

```
char     record[32];
char     output[128];
int      outputlen ;

/*
```

```
 * Create the first field as a string followed
 * by a field delimiter, taken from a 4
 * byte integer.
 */
outputlen = sprintf(output,
              "%d\376",*((int*)&record[0]));
/*
 * Now add the second field to the output record.
 * This is a 20 character field.
 */
memcpy(&output[outputlen],&record[4],20);
/*
 * Finally add the field delimiter for the
 * second field, and create the third field taken
 * from a floating point number.
 */
outputlen += (20 + sprintf(
              &output[outputlen+20],
              "\376%f",*((float*)&record[24]));
```

This conversion process can specifically be bypassed by the calling application if required. This may be needed from time to time for certain applications to read in the data in binary format without any conversions. The read record function should check the integer at FileDescriptor->ConvertFlags to see what sort of conversion should take place (see the OPEN and IOCTL functions).
The example below shows a simple read record process that simply returns 4 fields in a record, each field delimited by 0xfe (or 0376 in octal). No account of any record locks is made, no account of reading single fields is allowed for and no account of any conversions is used.

```
char    temprecord[4096];
int     temprecordlen ;
int     linecount ;
/*
 * Create the record, delimited by 0xfe
 * (or 376 in octal), in a temporary automatic
 * variable. This record will just be the name of
 * the record key copied 4 times. This means
 * we will have 3 field delimiters.
 */
for (temprecordlen = linecount = 0 ;
            linecount < 4 ; linecount++)
{
```

```
    memcpy(&temprecord[temprecordlen],
           RecordKey,RecordKeyLen);
    if (linecount == 3)
    {
        temprecordlen += RecordKeyLen ;
    }
    else
    {
        temprecord[temprecordlen+RecordKeyLen]
             = 0376 ;
        temprecordlen += (RecordKeyLen +1) ;
    }
}
/*
 * Make sure the caller has provided enough space
 * for the data to be copied to. If not, then
 * allocate more space.
 */
if (temprecordlen > (*BufferLenPtr)) {
    if (((*BufferPtr) = JediReadMalloc(
         FileDescriptor,temprecordlen)) == NULL)
    {
        return errno;
    }
}
/*
 * Copy the data to the users buffer and return to
 * the caller the length of the record returned.
 */
memcpy(*BufferPtr,temprecord,temprecordlen);
*BufferLenPtr = temprecordlen ;
   return 0 ;
```

NOTE: that when the buffer was allocated, instead of using JediMalloc() we actually used JediReadMalloc(). This is a special case for performance reasons and the database driver should use a call to JediReadMalloc() only when it is creating a buffer for data to be read into. There are no other occasions when you use JediReadMalloc(). The application can make use of this by trapping calls to JediReadMalloc(), as it knows that during these calls the database driver is creating a data space to return the record. Thus, the application can make sure the data is read into the final resting place for the data, rather than into some temporary buffer allocated created by JediMalloc(), that then needs copying elsewhere.

There are no locking flags passed to the READ function. The is because the application request to read a record with a lock goes through some base code in jEDI , and before the request reaches the database driver, jEDI will make a call to the LOCK function (detailed later) to do any record locking. Hence, there is no requirement for the READ functionality in the database driver to explicitly worry about locks.

The database driver may support the conversion of data, for example when reading in UNIX files it may convert new-line characters to attribute marks (from 0x0a to 0xfe). If this is supported by the database driver, the ConvertFlags member in the FileDescriptor structure passed to this function will show what sort of conversion, if any, to perform. This ConvertFlags member is a number of bits defined by JEDI_CONVERT_xxx in the header file jedi.h , and is initialised during the OPEN call. Subsequent calls by the application to IOCTL can modify this field. It is up to the database driver to decide if to support this functionality or not.

WRITE: Write a record to a file.

This function is called when the application wants to write a record to the file.

SYNOPSIS:

```
static  int WRITE(FileDescriptor , Flags ,
                  RecordKey ,RecordKeyLen ,
                  BufferPtr , BufferLen,
                  FieldNumber)
JediFileDescriptor  * FileDescriptor;
int                   Flags ;
char                * RecordKey ;
int                   RecordKeyLen ;
char                * BufferPtr ;
int                   BufferLen ;
int                   FieldNumber ;
```

PARAMETERS:

FileDescriptor. (Input parameter) This is the file descriptor that was returned when the file was originally opened.
Flags. (Input parameter) One or more of the following bits

JEDI_RECORD_FIELDREADWRITE. If set, then the value at parameter ‘FieldNumber’ contains the field number (or attribute) to write, instead of the entire record. If the database drive doesn’t support the writing of individual fields, then it should have not set the JEDI_STATUS_FIELDWRITE bit during the OPEN function (see the OPEN function for details of the ‘Status’ member of JediFileDescriptor) and this can be ignored.

RecordKey. (Input parameter) This points to an array of characters that define the record key for which to write the record as. The array is not a 0 terminated string.
RecordKeyLen. (Input parameter) Shows the length of the record key as pointed to by the parameter RecordKey.
BufferPtr. (Input parameter) This points to the actual data to write out, and is of length BufferLen bytes.
BufferLen. (Input parameter) This shows the amount of data to be written, as pointed to by the BufferPtr parameter.
FieldNumber. (Input parameter) If the caller wishes to write an individual field instead of the entire record, then they will set the JEDI_RECORD_FIELDREADWRITE bit in the Flags parameter and set this parameter to be the field number to write out. If the database driver does not support this

operation, you can ignore this field (see earlier description of JEDI_RECORD_FIELDREADWRITE).

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

OPERATION:

The WRITE function should, by default, write out an entire record. If the bit defined by (Flags & JEDI_RECORD_FIELDREADWRITE) is set, then the database driver should just write a single field, as given by parameter FieldNumber. Note that if database driver doesn't want to support writing of individual fields, then it should not set the JEDI_STATUS_FIELDWRITE bit in the Status field of the file descriptor during the OPEN function call, leaving it up to the application to insert the individual field and write the entire record.
There are no locking flags passed to the WRITE function. The is because the application request to write a record and maintain or release a lock goes through some base code in jEDI, and before the request reaches the database driver, jEDI will make a call to the LOCK function (detailed later) to do any record locking releases. Hence, there is no requirement for the WRITE functionality in the database driver to explicitly worry about locks.
The foreign database accessed by the database driver will probably have an entirely different record structure to that expected by the application. Therefore it is up to the database driver to convert the record from the format by the application to the format of the foreign database. This is the reverse process as detailed in the READ record function described previously. The format of the record that is pointed to by the BufferPtr parameter is that whereby all the fields in the record are string fields delimited by a 0xfe (or octal 0376) characters. Like the READ record function, the database driver has to convert the fields as necessary.
This conversion process can specifically be bypassed by the calling application if required. This may be needed from time to time for certain applications to write out the data in binary format without any conversions. The write record function should check the integer at FileDescriptor->ConvertFlags to see what sort of conversion should take place (see the OPEN and IOCTL functions).
The example below shows a simple write record process that simply writes the record out to a UNIX flat file. No account of writing single fields is allowed for and no account of any conversions is used.

```
char    ActualPathName[PATH_MAX+1];
int             i1 , fd1  ;
/*
 * Make up the name of a UNIX file to write to.
 * This is basically the concatenation of the file
 * name that was opened plus the record key.
 */
```

```
i1 = strlen(FileDescriptor->PathName);
memcpy(ActualPathName,FileDescriptor->PathName,i1);
ActualPathName[i1] = '/';
memcpy(&ActualPathName[i1+1],RecordKey,
        RecordKeyLen);
ActualPathName[i1+1+RecordKeyLen] = NULL;
/*
 * Now to try to open the file.
 */
if ((fd1=open(ActualPathName,
           O_WRONLY|O_TRUNC|O_CREAT,0666)) < 0)
{
    return errno;
}
/*
 * Now to write the data to the actual file.
 */
if (write(fd1,BufferPtr,BufferLen) != BufferLen)
{
    close(fd1);
    return (errno == 0 ? EIO : errno);
}
/*
 * Write succeeded. Close the file and return 0.
 */
close(fd1);
return 0 ;
```

The database driver may support the conversion of data, for example when reading in UNIX files it may convert new-line characters to attribute marks (from 0x0a to 0xfe). If this is supported by the database driver, the ConvertFlags member in the FileDescriptor structure passed to this function will show what sort of conversion, if any, to perform. This ConvertFlags member is a number of bits defined by JEDI_CONVERT_xxx in the header file jedi.h, and is initialised during the OPEN call. Subsequent calls by the application to IOCTL can modify this field. It is up to the database driver to decide if to support this functionality or not.

DELETE: Delete a Record from a File.

This function is called when the application wants to delete a record from the file.

SYNOPSIS:

```
static  int DELETE(FileDescriptor , Flags ,
                   RecordKey , RecordKeyLen )
JediFileDescriptor   * FileDescriptor;
int                    Flags ;
char                 * RecordKey ;
int                    RecordKeyLen ;
```

PARAMETERS:

FileDescriptor. (Input parameter) This is the file descriptor that was returned when the file was originally opened.

Flags. (Input parameter) This parameter not used at present.

RecordKey. (Input parameter) Points to a string describing the record key to delete Note the record key is not a 0 terminated string.

RecordKeyLen. (Input parameter) This is the length of the record key as passed by RecordKey.

RETURN VALUE:

0 is returned if the record is deleted successfully.

ENOENT is returned if the record did not exist.

Any other value shows the reason the operation failed using the values defined in errno.h

OPERATION:

The DELETE function simply removes the record from the database. The following is a very simple example of deleting a record, where we have assumed the database is simply a UNIX flat file.

```
char    ActualPathName[PATH_MAX+1];
int             i1 , fd1  ;
/*
 * Make up the name of a UNIX file to delete.
 */
i1 = strlen(FileDescriptor->PathName);
memcpy(ActualPathName,FileDescriptor->PathName,i1);
ActualPathName[i1] = '/';
memcpy(&ActualPathName[i1+1],RecordKey,
        RecordKeyLen);
ActualPathName[i1+1+RecordKeyLen] = NULL;
```

```
/*
 * Now to delete the actual record.
 */
if (unlink(ActualPathName) != 0)
{
    return (errno == 0 ? ENOENT : errno);
}
return 0 ;
```

There are no locking flags passed to the DELETE function. The is because the application request to delete a record goes through some base code in jEDI, and before the request reaches the database driver, jEDI will make a call to the LOCK function (detailed later) to release any record locking. Hence, there is no requirement for the DELETE functionality in the database driver to explicitly worry about locks.

CLEAR: Delete All Records from a File.

This function is called by an application to delete all the records from a file. No file locks are taken, so other applications could be writing to the file at the same time.

SYNOPSIS:

```
static  int CLEAR(FileDescriptor)
JediFileDescriptor  *FileDescriptor;
```

PARAMETERS:

FileDescriptor. (Input parameter) This is the file descriptor that was returned when the file was originally opened.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

OPERATION:

The database driver simply clears all records from the file in the most efficient manner it can. It is up to the database driver to decide to return an error code if some records cannot be deleted. For example, in the driver supplied with jBASE to use UNIX files as a database, if some records cannot be deleted (i.e. the UNIX file cannot be unlinked), for example because of file permissions, then no error is reported. However, the database driver is free to choose whether to ignore such failures or not. The following crude example shows how a database drive that uses the UNIX files as a database might delete all records (i.e. UNIX files).

```
DIR      *DirPtr;
struct   dirent *Next;
char     PathName[PATH_MAX*2];
struct   stat stat_info ;

if ((DirPtr = opendir(FileDescriptor->PathName))
                == NULL)
{
    return errno;
}
while ((Next = readdir(DirPtr)) != NULL)
{
    if ((ReturnValue=JediBaseSignalCheck()) != 0)
```

```
    {
        break;
    }
    strcpy(PathName,FileDescriptor->PathName);
    strcat(PathName,"/");
    strcat(PathName,Next->d_name);
    /*
     * Find out the type of the entry. We will not
     * attempt to delete anything other than
     * regular files.
     */
    if (stat(PathName , &stat_info) == 0)
    {
        if (stat_info.st_mode & S_IFREG)
        {
            unlink(PathName);
        }
    }
}
closedir(DirPtr);
return 0;
```

NOTE: the function call to JediBaseSignalCheck(). This function will return a non-zero value should the user decide to interrupt and abort the clear file operation. This function is documented earlier in the chapter.

LOCK: Provide Record Locking Mechanism.

This function is called by the application to provide record locking support. It can also be called intrinsically via the jEDI code to support record locking functions. For example, if an application performs a request to READ a record with a lock, then the jEDI function will call the LOCK function directly, rather than the READ function doing the call itself.

SYNOPSIS:

```
static  int LOCK(FileDescriptor , Flags ,
                 RecordKey , RecordKeyLen )
 JediFileDescriptor     * FileDescriptor;
int                       Flags ;
char                    * RecordKey ;
int                       RecordKeyLen ;
```

PARAMETERS:

FileDescriptor. (Input parameter) This is the file descriptor that was returned when the file was originally opened.
Flags. (Input parameter) Can be one of the following bits to show what operation is to be performed by the LOCK function:

JEDI_RECORD_LOCKRECORD Take a record lock and wait.
JEDI_RECORD_LOCKRECORD_NOWAIT Take a record lock, but return if lock already taken (The function return code is JEDI_ERRNO_LOCK_TAKEN).
JEDI_RECORD_UNLOCKRECORD Remove a record lock.

RecordKey. (Input parameter) This is a pointer to a character array, which shows what record key is to be locked or unlocked. Note that this array is not a 0 terminated string (See RecordKeyLen parameter).
A special case exists for when NULL is passed in this field. This means the calling application wants to release all the locks for all the records in the file held by this application.
RecordKeyLen. (Input parameter) This is the length of data passed in the RecordKey parameter.

RETURN VALUE:

0 shows the locks were created/removed okay.
JEDI_ERRNO_LOCK_TAKEN shows the JEDI_RECORD_LOCKRECORD_NOWAIT bit was set, and the record/field was already locked by another process.
EDEADLK shows one of the lock bits was set, avoiding a damaging

Any other value shows the reason the operation failed using the values defined in errno.h.

OPERATION:

The function should check if RecordKey is NULL. If it is, then release all locks held by this application for this file. If the RecordKey is not NULL, then the function should either take or release an individual record key lock.

A locking function is supplied with jBASE called JediSystemLock that can be used to provide a record locking scheme. It is not a 100% genuine record locking scheme, but a scheme whereby a pseudo-random integer value is created from the record key and the pseudo-random value is the value that is locked. While not a 100% record locking scheme, the chances of a clash whereby different record keys generate the same pseudo-random value are very small.

The following shows how the LOCK function could be written using the supplied JediSystemLock() function. However, the database driver is free to use alternate locking mechanisms if required. In this case the LOCK function will probably become an interface to the record locking scheme provided by the foreign database.

```
unsigned  int  HashValue ;
unsigned  char c1 , c2 ;
/*
 * First of all, create a HASH value for
 * the RecordKey.
 */
if (RecordKey != NULL)
{
    HashValue = JediBaseHash(RecordKey ,
                  RecordKeyLen , 1);
    /*
     * Mask off the top bit to make it a signed
     * positive value. Due to bug in a previous
     * compiler, we mask the top two bits.
     */
    HashValue &= 0x3fffffff;
}
/*
 * Secondly, see if we are to remove locks.
 */
if (Flags & JEDI_RECORD_UNLOCKRECORD)
{
    if (RecordKey == NULL)
    {
        /*
```

```
         * Release ALL locks on the file.
         */
        return JediSystemLock(FileDescriptor ,
                     JEDI_UNLOCK_ALL , 0) ;
    }
    /*
     * Release a single lock on the file.
     */
    return JediSystemLock(FileDescriptor ,
                  JEDI_UNLOCK , HashValue ) ;
}
/*
 * Thirdly, we must be setting a lock. Do we have
 * to wait or not ?
 */
return JediSystemLock(FileDescriptor ,
         (Flags & JEDI_RECORD_LOCKRECORD_NOWAIT ?
         JEDI_LOCK_NOWAIT : JEDI_LOCK ),
             HashValue ) ;
```

IOCTL: Support database driver control functions.

This function is called to provide general database driver operations. Like the ioctl() function in 'C' programs, there are some standard commands that most database drivers will be expected to perform, some that are optional, and some that are unique to the database driver.

SYNOPSIS:

```
static  int IOCTL(FileDescriptor , SubCommand ,
                  IoctlAddr , IoctlLen ,
                  IoctlReturnAddr , IoctlReturnLen)
JediFileDescriptor  * FileDescriptor;
int                   SubCommand  ;
void                * IoctlAddr ;
int                   IoctlLen ;
void                * IoctlReturnAddr ;
int                 * IoctlReturnLen;
```

PARAMETERS:

FileDescriptor. (Input parameter) This is the file descriptor that was returned when the file was originally opened.
SubCommand. (Input parameter) This shows actually what sort of operation to perform. There are some common commands given by the JEDI_IOCTL_xxx definition, but the database driver can add support for other commands if required.
IoctlAddr. (Input parameter) This gives the address of any optional buffer that may be necessary to provide extra specification depending on the SubCommand. Therefore its use is highly dependent upon SubCommand. The application may pass NULL here.
IoctlLen. (Input parameter) This is the size of data that is pointed to by the IoctlAddr parameter.
IoctlReturnAddr. (Input and Output parameter) This is the address of a buffer or length IoctlReturnLen bytes where the IOCTL function can return any data depending upon the command to be executed (See SubCommand).
IoctlReturnLen. (Input and Output parameter) This is the address of an integer that shows the length of the return buffer pointed to by IoctlReturnAddr. Upon exit, the database driver should update this to show the amount of valid data that the IOCTL function has returned in IoctlReturnAddr.

RETURN VALUE:

The return value is highly dependent on the database driver. By convention though you should return -1 if the operation failed, 0 if the operation was successful. In the example code given, if you decide to include the standard IOCTL functions, then return the same return code as given in the example code.

OPERATION:

The use of IOCTL is highly database driver dependent. However, there are some common functions that a database driver may be expected to perform. The code example below shows all these common functions. All the functionality is optional, and all the jBC utilities that make IOCTL calls have suitable defaults should the database driver decide not to support the command.

```
int       ReturnValue;
int       i1 , ReturnBufferLen ;
char      ReturnBuffer[4096];
char      FullName[4096];
char      *Ptr1 , *Ptr2 ;
struct    stat  RecStat ;
char      ActualPathName[PATH_MAX+1];
int       SaveErrno ;
int       fd1 ;

ReturnValue = 0;
switch(SubCommand)
{
case  JEDI_IOCTL_CONVERT:
    /*
     * The application wants to change the status
     * of the conversion flags, which are usually
     * examined to see if to read a record in
     * binary format without conversion, or to do
     * the conversion between fields on the foreign
     * database and the format expected by the
     * application.
     * The input string will be something like
     * “RB,WB”, which means all the reads should
     * be binary and the writes should be binary.
     */
    if (IoctlReturnAddr != NULL &&
            IoctlAddr != NULL)
    {
        FileDescriptor->ConvertFlags =
          JediBaseIoctlConvert(
             FileDescriptor->ConvertFlags ,
             IoctlAddr , IoctlLen ,
             IoctlReturnAddr , IoctlReturnLen) ;
```

```
    }
    break;
case  JEDI_IOCTL_FILESTATUS:
    /*
     ** The application wants details of the file.
     *
     * Return details as an attribute delimited
     * record as follows :
     * <1>    File Type as a string
     * <2>    FileFlags , as decimal number ,
     *        showing the LOG, BACKUP and
     *        TRANSACTION permissions.
     * <3>    BucketQty , as decimal number ,
     *        number of buckets in the file.
     *        Not applicable in this example.
     * <4>    BucketSize , as decimal number , size
     *        of each bucket in bytes.
     *        Not applicable.
     * <5>    SecSize , as decimal number , size of
     *        secondary data space.
     *        Not applicable in this example.
     * <6>    Restore Spec , a string showing any
     *        restore re-size specification.
     *        Not applicable in this example.
     * <7>    Locking identifiers, delimited by
     *        multi-values.
     * <8>    FileFlags  showing LOG, BACKUP and
     *        TRANSACTION permissions. Multi-values
     *        are :
     *        <8,1> set to 1 if no logging.
     *        <8,2> set to 1 if no transaction.
     *        <8,3> set to 1 if no file backup.
     */
    ReturnBufferLen =
       sprintf(ReturnBuffer,
              "Example\3760\376\376\376\376\376%d
              \375%d\3760\3750\3750",
              FileDescriptor-LockId1,
              FileDescriptor->LockId2);
    if (IoctlReturnAddr != NULL)
    {
```

```
        i1 = min(ReturnBufferLen,*IoctlReturnLen);
        memcpy(IoctlReturnAddr,ReturnBuffer,i1);
        *IoctlReturnLen = i1 ;
    }
    else
    {
        ReturnValue = EINVAL;
    }
    break;
case JEDI_IOCTL_HASH_LOCK:
    /*
      * Find the HASH value used in any locking
      * scheme.
      */
     if (IoctlReturnAddr != NULL &&
         IoctlAddr != NULL)
    {
        i1 = JediBaseHash(IoctlAddr , IoctlLen ,1);
        /*
         * Mask off the top bit to make it a
         * signed positive value.
         * Due to bug in a previous compiler,
         * we mask the top two bits.
         */
        i1 &= 0x3fffffff;
        *IoctlReturnLen =
              sprintf(IoctlReturnAddr,"%d",i1);
    }
    break;
case JEDI_IOCTL_FINDRECORD:
    /*
     * Test whether record exists.
     * Work out a full path name to use.
     */
    i1 = strlen(FileDescriptor->PathName);
    memcpy(ActualPathName,
           FileDescriptor->PathName,i1);
    ActualPathName[i1] = '/';
    memcpy(&ActualPathName[i1+1],I
             IoctlAddr,IoctlLen);
    ActualPathName[i1+1+Ioctl] = NULL;
```

```
    if ((fd1=open(ActualPathName,O_RDONLY)) < 0)
    {
        ReturnValue = errno;
    }
    close(fd1);
    if ( IoctlReturnAddr != NULL )
    {
        i1 = min(IoctlLen,*IoctlReturnLen);
        memcpy(IoctlReturnAddr,IoctlAddr,i1);
        *IoctlReturnLen = i1 ;
    }
    break;
default:
    ReturnValue = -1;
    break;
}
return ReturnValue ;
```

SYNC: Synchronize the Data to Disk.

This function is called to flush any cache data to the disk. This is quite specialized and if necessary the database driver can choose to do nothing except give a return code of 0.

SYNOPSIS:

```
static  int SYNC(FileDescriptor)
JediFileDescriptor  *FileDescriptor;
```

PARAMETERS:

FileDescriptor. (Input parameter) This is the file descriptor that was returned when the file was originally opened.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

OPERATION:

The following code example shows the use of the UNIX fsync() function to provide this functionality. Note that not all derivatives of UNIX support fsync().

```
errno = 0;
if (fsync(FileDescriptor->FileFd) == 0) return(0);
    return errno;
```

JEDI API calls

The previous chapter concentrated on creating new database drivers, enabling existing jBC or C programs to access alternative databases. The most common uses of jEDI will be through statements such as READ, WRITE, and OPEN etc. within a jBC application.
JEDI is designed to be a completely autonomous component of jBASE, and while a jBC program uses it extensively, it is very capably of being used by a C program that has no jBC code anywhere in sight.
The purpose of this section is to describe how to write such a C program that calls the jEDI API. Many of the tools provided with jBASE use jEDI in this manner.
An example source of a C program calling jEDI directly is provided with jBASE in the file $JBCRELEASEDIR/src/jediCExample.c. This program performs a simple file copy from one jEDI supported file to another jEDI supported files, and shows the use of many of the functionality described later.
Provided with jBASE are a number of #include files you will need in your program. These files are jsystem.h and jedi.h and are found in directory $JBCRELEASEDIR/include. All the non-standard definitions described in the section, such as JEDI_TRANSLOG_COMMAND_QUERY are defined in one of these two files, so you should #include both of them. Note that jsystem.h also does a '#include <stdio.h>'.
The stages of calling jEDI directly from a C program can be summarised as:

- Initialisation.
- Making jEDI database requests.
- Program termination.
-

The operations of transaction boundaries and transaction journaling are done automatically be jEDI and no intervention on the part of the C program is required, with the exception of abnormal program termination, which is discussed later.

Initialisation.

There are two operations to perform before making the first database operation, both of which are optional.

Set up signal handlers. You should ideally set up signal handlers such that in the event of an unexpected program termination, you can still perform the wrapup operations detailed in ‘Program Termination’ later. Basically, at program termination you will abort any transaction and close all opened files (thus releasing all the locks). Most database drivers, including the ones supplied with jBASE, can copy with abnormal exits that do not wrapup properly. Thus, while setting up signal handlers and wrapping up cleanly is the preferred option, it should not be mandatory.

Connect to the database. This involves calling the function ‘JediConnect’ to return a jEDI connection handle. This connection handle is then passed to all subsequent OPEN requests. If this stage is omitted, then you can always use the default connect handle of 0. However, this function provides definitions for storage allocation functions to call (e.g. malloc, strdup), which may provide performance improvements in your code. A typical call to JediConnect might be:

```
static  void * readalloc() ;
int     ConnectHandle ;
struct  JediConnectBlock ConnectBlock ;
/*
 * Set up the database name to connect to.
 * This is reserved at present, so use “” string.
 */
ConnectBlock.dbname = "" ;
/*
 * Set up the addresses of all the memory
 * allocation functions we shall require.
 * Note the ‘readmallocptr’ member, which is to
 * our own allocation function called ‘readalloc’
 * As we see later, this will provide performance
 * improvements.
 */
ConnectBlock.mallocptr = (void *(*)())malloc ;
ConnectBlock.readmallocptr=(void *(*)())readalloc;
ConnectBlock.reallocptr = (void *(*)())realloc ;
ConnectBlock.freeptr = free ;
ConnectBlock.strdupptr = (char *(*)())strdup ;
/*
 * Connect to jedi and check return code.
 */
if ((ConnectHandle=JediConnect(&ConnectBlock))<0)
```

```
{
    perror("JediConnect");
    exit(1);
}
/*
 * We use the variable ConnectHandle in all
 * subsequent calls to jedi.
 */
```

Making jEDI database requests.

Once any initializations have been performed, then you can start calling jEDI API functions. The first call is usually to JediOpen, followed by a number of requests using the returned file descriptor.
One of the limitations during this phase is the C program MUST keep the environment variable PWD up to date. The variable PWD is initialised to the current directory by jEDI when the first ever JediOpen function call is made. However, should the program change the directory using calls to ‘chdir then they must also update the PWD directory.
For clarity, a full description of these function calls is given later, and a summary now follows.

- **JediOpen**. Open a file.
- **JediOpenDeferred**. Open a file in deferred mode, i.e. it is only partially opened, and will be genuinely opened on the first database access using the returned file descriptor.
- **JediClose**. Close a file.
- **JediSelect**. Initiate a selection of a list of record keys.
- **JediSelectEnd**. Terminate the selection of the list of record keys.
- **JediReadnext**. Read the next record key following a selection.
- **JediReadRecord**. Read a record (or a field from within a record) from the file.
- **JediWriteRecord**. Write a record (or a file from within a record) to a file.
- **JediDeleteRecord**. Delete an entire record.
- **JediLock**. Perform record locking.
- **JediIOCTL**. General-purpose file control.
- **JediDeleteFile**. Delete a file.
- **JediSync**. Flush updates to the journaler and to disk.
- **JediClearFile**. Clear all records from the file.

Program termination.

There are basically two things to do when a program terminates, either normally or because of a signal. You should first abort any outstanding transactions (assuming you have used transaction support), and secondly close any opened file.

Aborting outstanding transaction. If you are using transaction boundaries in your program, you should check to see if you have an open transaction, and handle appropriately to your application. The following code shows a check to see if a transaction was open, and if so the transaction is aborted and an error message is printed.

```
if (JediTransLog(JEDI_TRANSLOG_COMMAND_QUERY))
{
    JediTransLog(JEDI_TRANSLOG_COMMAND_ABORT , 0 ,
        “Abort from program xxx” );
    fprintf(stderr,”Forced transaction abort\n”);
}
```

Close opened files. You should ideally keep a list of all currently opened files so that you can close them at a later stage. The closing of files will also release all the locks on the file.

Compiling and linking a program

Once you have written your C application, you need to compile it and link it. There are two considerations here. Firstly, where to find the #include files jsystem.h and jedi.h Secondly, where to find the jedi library.

Finding the #include files.

If you use the cc command to build your application, then you need to specify on the command line where the include files are, for example:

```
% cc -c myprog.c -I $JBCRELEASEDIR/include
```

If you use the jbc command to build your application, then the command is simplified, for example:

```
% jbc -c myprog.c
```

Linking with the jEDI library.

If you use the cc command to build your application, then you need to specify in which library the jEDI API functions can be found. There are a number of ways to achieve this. The following examples may vary slightly with different UNIX development environments. The first example shows linking with shared objects using the cc command:

```
% cc myprog.o -o myprog -L $JBCRELEASEDIR/lib
          -l jedi -l ld
```

The next example shows linking with archive libraries:

```
% cc myprog.o -o myprog $JBCRELEASEDIR/lib/libjedi.a
        -l ld
```

If you use the jbc command to build your application, then the command is simplified. There are basically two ways of building your application. The first is the normal, using jEDI-shared libraries. The second uses jEDI archive libraries. For example, to link the object to create an executable program using the jbc command and shared libraries:

```
% jbc myprog.o -o myprog
```

To link with archive libraries, the command would become:

```
% jbc myprog.o -o myprog -Jla
```

When linked with jEDI-shared libraries, you must ensure the environment variable LD_LIBRARY_PATH shows where it can find these shared libraries (use LIBPATH for AIX systems). For example, you should do this in your. Profile:

```
export LD_LIBRARY_PATH=$JBCRELEASEDIR/lib
```

Transaction boundary support

This can be provided within a C program using the calls to the supplied function JediTransLog. There are four calls - start, end (commit), abort (rollback) and query.

Transactions cannot be nested, so if you start a transaction when a transaction is already active, then an error code is returned.

Starting a transaction:

```
if (JediTransLog(JEDI_TRANSLOG_COMMAND_START , Flags ,
          “message”) != 0)
{
    perror(“TRANSTART”);
}
```

Where:

- **JEDI_TRANSLOG_COMMAND_START**: Show the function to start a transaction boundary.
- **Flags**. If the bit JEDI_TRANSLOG_FLAGS_SYNC is set, then following the termination of the transaction though an abort or commit, the updates will be flushed to disk and if the transaction journaling is operative, the update information flushed to the output media. This gives greater database integrity in the event of a failure at the cost of an additional overhead. If no synchronisation is required, pass 0 in this parameter.
- **“message”**. Can be any 0 terminated string of characters, and is used simply by the transaction journaler to record information about the transaction. Hence, this string should be meaningful should the updates wish to be later examined though the transaction journaling mechanism.

The return value is 0 for successful, and any other value is an error.

Ending (committing) a transaction :

```
if (JediTransLog(JEDI_TRANSLOG_COMMAND_END , Flags ,
          “message”) != 0)
{
    perror(“TRANSEND”);
}
```

Where:

- **JEDI_TRANSLOG_COMMAND_END**: Show the function to end or commit a transaction.
- **Flags**. Not used with this function call.
- **“message”**. Can be any 0 terminated string of characters, and is used simply by the transaction journaler to record information about the transaction. Hence, this string should be meaningful should the updates wish to be later examined though the transaction journaling mechanism.

The return value is 0 for successful, and any other value is an error.

Aborting (rollback) a transaction :

```
if (JediTransLog(JEDI_TRANSLOG_COMMAND_ABORT , Flags ,
        “message”) != 0)
{
    perror(“TRANSABORT”);
}
```

Where:

- **JEDI_TRANSLOG_COMMAND_ABORT**: Show the function to abort or rollback a transaction.
- **Flags**. Not used with this function call.
- **“message”**. Can be any 0 terminated string of characters, and is used simply by the transaction journaler to record information about the transaction. Hence, this string should be meaningful should the updates wish to be later examined though the transaction journaling mechanism.

The return value is 0 for successful, and any other value is an error.

Query the status of a transaction :

```
if (JediTransLog(JEDI_TRANSLOG_COMMAND_QUERY))
{
    printf(“A transaction is active\n”);
}
else
{
    printf(“No transaction open\n”);
}
```

Where:

- **JEDI_TRANSLOG_COMMAND_QUERY**: Show the function to return a zero if process not inside a transaction, or non-zero if process i.

The return value is 0 if the process is not in a transaction, or non-zero if the process is inside a transaction.

jEDI Environment variables

When using a program with jEDI , you should be aware of some of the environment variables used by the jEDI library :

PWD. This will be initialised by jEDI following the first call to JediOpen. Once set up by jEDI , the program must keep this up to date should the program call chdir() at any point to change the current working directory.

HOME. This defines the home directory of the logged on user. It is used by jEDI when resolving Q pointers, which point to an alternative user name.

LD_LIBRARY_PATH. Used by most UNIX systems to define the path to look for shared objects. If you write a C program that was linked with jEDI shared objects, then before the program can be executed you must set LD_LIBRARY_PATH to include, as one of its components, the name of the directory where it can find the jEDI shared object.

LIBPATH. Same as LD_LIBRARY_PATH, but for AIX systems

JEDIFILENAME_MD. This defines if you have a Master Dictionary file in operation. If so, then during a call to JediOpen the function will look in this file to see if the file is defined here. If a record exists in the file defined by JEDIFILENAME_MD, and the first field is Q, then this is a Q pointer and is resolved as such.

JEDIFILENAME_SYSTEM. If this variable is set, then it is used by jEDI to find out more about pseudo jBASE account names during Q pointer resolution.

JEDIFILEPATH. When a file is opened, it is usually by a relative name such as “CUSTOMERS” or “PIPES”. This variable can contain a number of directory names, delimited by a colon, to describe the search path to look for the file. For example, if the environment variable JEDIFILEPATH contained “/home/greg:/home/fat” then jEDI would first try to open file “/home/greg/CUSTOMERS” followed by “/home/fat/CUSTOMERS”. If this variable is not defined, then jEDI defaults to the home directory followed by the current directory.

JBCOBJECTLIST. This is used when attempting to load user-written database drivers. It gives a list of directories where the shared objects may exist

JediOpen: Open a File.

This function is called whenever the application requires a file to be opened.

SYNOPSIS:

```
Int  JediOpen(ConnectHandle , FilePointer,
                PathName , FilePath );
int                    ConnectHandle ;
JediFileDescriptor  ** FilePointer ;
char                 * PathName ;
char                 * FilePath ;
```

PARAMETERS:

ConnectHandle. This is the value passed back by the call to JediConnect, which was documented earlier in this chapter. If the call to JediConnect was omitted, then simply pass 0 in this parameter.
FilePointer. The address of a file descriptor pointer if the open is successful, then we return here the file descriptor this file descriptor is passed on all subsequent jEDI API calls to define the opened file.
PathName. This is the name of the file to open, and can be in a number of guises. Examples are:

- **"CUSTOMERS"**. The file name to open is CUSTOMERS, and the open function will make use of the 'FilePath' parameter to establish the directory that CUSTOMERS is contained in.
- **"./CUSTOMERS"**. The file name to open is CUSTOMERS, but the use of the preceding ./ means we ignore the 'FilePath' parameter and just look in the current directory.
- **"../CUSTOMERS]D"**. Open the dictionary section of file CUSTOMERS, which will have a UNIX file name of CUSTOMERS]D. The use of the preceding ../ means we ignore the 'FilePath' parameter and just look in the parent directory.
- **"DICT PROSPECTS"**. Open the dictionary section of file PROSPECTS and the open function will make use of the 'FilePath' parameter to establish the directory that CUSTOMERS is contained in.
- **"/home2/accts/ORDERS,1995"**. Open the section 1995 of the file ORDERS, which is in directory /home2/accts. The 'FilePath' parameter will not be used.

FilePath. If a simple file name is given to be opened, without a leading "/" or "./" or "./" then we use this parameter to establish what directories to look in. A colon delimits each directory name in this parameter. You would normally pass NULL here, and JediOpen uses the environment variable JEDIFILEPATH instead. Note that if FilePath is NULL , and JEDIFILEPATH is undefined, then we search for the file in the users home directory followed by the current working directory.

RETURN VALUE:

0 shows the file was opened successfully.
ENOENT shows the file could not be opened.
Any other value shows the reason the operation failed using the values defined in errno.h.

JediOpenDeferred: Deferred open.

This function is called whenever the application requires a file to be opened. It is similar to the JediOpen function call, except that the file is only pseudo-opened. In fact, the function will almost always succeed.

The first database operation performed on the file descriptor will cause the file to be actually opened. If this open fails, then an appropriate error return code is returned to the caller of the database operation.

The use of this function is highly specialized. One use of it is where, as part of a global common initialisation routine, you open all the files you may ever use. This is a common technique in application programming. By calling JediOpenDeferred you have a much lower start-up overhead, and only incur the overhead of a real open when that file is actually used.

SYNOPSIS:

```
int  JediOpenDeferred(ConnectHandle , FilePointer,
                PathName , FilePath );
int                    ConnectHandle ;
JediFileDescriptor  ** FilePointer ;
char                 * PathName ;
char                 * FilePath ;
```

PARAMETERS:

ConnectHandle. This is the value passed back by the call to JediConnect, which was documented earlier in this chapter. If the call to JediConnect was omitted, then simply pass 0 in this parameter.

FilePointer. The address of a file descriptor pointer if the open is successful, then we return here the file descriptor this file descriptor is passed on all subsequent jEDI API calls to define the opened file.

PathName. This is the name of the file to open, and can be in a number of guises. Examples are:

- **“CUSTOMERS”**. The file name to open is CUSTOMERS, and the open function will make use of the ‘FilePath’ parameter to establish the directory that CUSTOMERS is contained in.
- **“./CUSTOMERS”**. The file name to open is CUSTOMERS, but the use of the preceding ./ means we ignore the ‘FilePath’ parameter and just look in the current directory.
- **“../CUSTOMERS]D”**. Open the dictionary section of file CUSTOMERS, which will have a UNIX file name of CUSTOMERS]D. The use of the preceding ../ means we ignore the ‘FilePath’ parameter and just look in the parent directory.
- **“DICT PROSPECTS”**. Open the dictionary section of file PROSPECTS and the open function will make use of the ‘FilePath’ parameter to establish the directory that CUSTOMERS is contained in.
- **“/home2/accts/ORDERS,1995”**. Open the section 1995 of the file ORDERS, which is in directory /home2/accts. The ‘FilePath’ parameter will not be used.

FilePath. If a simple file name is given to be opened, without a leading “/” or “./” or “./” then we use this parameter to establish what directories to look in. A colon delimits each directory name in this parameter. You would normally pass NULL here, and JediOpen uses the environment variable JEDIFILEPATH instead. Note that if FilePath is NULL , and JEDIFILEPATH is undefined, then we search for the file in the users home directory followed by the current working directory.

RETURN VALUE:

0 shows the file was opened successfully.
ENOENT shows the file could not be opened.
Any other value shows the reason the operation failed using the values defined in errno.h.

JediClose: Close an Opened File.

This function is called to close an opened file. Any locks that exist will be released.
If the file is currently part of a transaction, then the jEDI code will defer the close until the end of the transaction - no effort is required on the part of the application and should assume the file is really closed.

SYNOPSIS:

```
int  JediClose(FileDescriptor , Flags)
JediFileDescriptor  * FileDescriptor;
int                   Flags ;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.
Flags. This parameter is not used at present.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

JediSelect: Select Record Keys from a File.

This function is called when the user wants to perform some sort of selection of the file. At present, the select function is called to select all records in the file. However, calls from future programs may add selection criteria.

SYNOPSIS:

```
int JediSelect(FileDescriptor , SelectPtr ,
                     SelectDetails, Index )
JediFileDescriptor    *  FileDescriptor;
struct JediSelectPtr  ** SelectPtr;
char                  *  SelectDetails;
int                      Index ;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.
SelectPtr. The caller passes here the address of a “struct JediSelectPtr *” variable. If the SELECT works, then this function will return here the address of an allocated JediSelectPtr structure. This address will be passed to future calls to SELECTEND and READNEXT.
SelectDetails. This is a NULL terminated string describing the selection details. For example, to select all record keys that match the regular expression “^MJI.*PIPE” the caller will set this variable to point to the string “^MJI.*PIPE”. Note that none of the supplied database drivers as of release 1.9 support this feature. Future database driver may do so. Currently the language for jBC does not support this either, but again may do so in the future.
Index. For database drivers that support multiple indexed files, this shows the index number to perform the select against. As of release 1.9, the jBC language does not directly support multiple indexed databases. However, this may change in future releases.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h .

JediSelectEnd: Terminate a Selection Process.

This function is called when the selection process is terminated.

SYNOPSIS:

```
int JediSelectEnd(FileDescriptor , SelectPtr )
JediFileDescriptor         * FileDescriptor;
    struct JediSelectPtr       * SelectPtr;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.
SelectPtr. This is the address of the structure that was created during the call to the JediSelect function.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

JediReadnext: Get Next Record Key

This function is typically called many times following a call to the SELECT function. This function will return the next record that has been selected following a call to the SELECT function.

SYNOPSIS:

```
int JediReadnext(FileDescriptor , SelectPtr ,
                 RecordKeyPtr , RecordKeyLenPtr)
JediFileDescriptor      *  FileDescriptor;
struct JediSelectPtr    *  SelectPtr;
char                    ** RecordKeyPtr ;
int                     *  RecordKeyLenPtr ;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.
SelectPtr. This is the address of the structure that was created during the SELECT function.
RecordKeyPtr. The application passes in the parameter a pointer to a pointer to a buffer. This is where the next selected record key will be returned. If the buffer is not large enough (See RecordKeyLenPtr), then the JediReadnext function will allocate sufficient memory and return at RecordKeyPtr the address of the allocated memory. It is up to the calling application to detect if this has been performed, and to free any allocated memory.
RecordKeyLenPtr. This is the address of an integer that describes the length of the buffer given by the expression "*RecordKeyPtr". When we have selected the next record key and placed it in the buffer "*RecordKeyPtr", we return here the length of the record key. If there are no more record keys, then -1 is returned here.

RETURN VALUE:

0 if the operation succeeded. This includes when the list of record keys are exhausted (in this case, *RecordKeyLenPtr set to -1).
Any other value shows the reason the operation failed using the values defined in errno.h.

JediReadRecord: Read a Record from a File.

This function is called when the application wants to read a record from the opened file.

Synopsis:

```
int JediReadRecord(FileDescriptor, Flags,RecordKey,
                   RecordKeyLen , BufferPtr,
                   BufferLenPtr , FieldNumber)
JediFileDescriptor    *  FileDescriptor;
int                      Flags ;
char                  *  RecordKey ;
int                      RecordKeyLen ;
char                  ** BufferPtr ;
int                   *  BufferLenPtr ;
int                      FieldNumber ;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.

Flags. One or more of the following bits

- JEDI_RECORD_FIELDREADWRITE. If set, then the value at parameter 'FieldNumber' contains the field number (or attribute) to read, instead of the entire record. If the database drive doesn't support the reading of individual fields, then it will not set the JEDI_STATUS_FIELDREAD bit during the JediOpen function and this will be ignored.

RecordKey. Pointer to a character array describing the record key to read in the array does not need to be 0 terminated.

RecordKeyLen. The length of the RecordKey parameter

BufferPtr. This is the address of a character array where we will place the record data. The length of this is given by the BufferLenPtr parameter. Should this buffer not be large enough to accommodate the record, we will allocate more space and return here the address of the data space allocated. Thus the calling function can determine where the record was placed, and thus whether it was placed in the character array supplied to READ, or the character array allocated by READ. It is up to the calling function to free any allocated space if necessary.

BufferLenPtr. This is the address of an integer, originally set up to describe the size of the character array pointed to by '* BufferPtr'. Upon return from READ, this will indicate the size of the record that was read in.

FieldNumber. If the caller wishes to read an individual field instead of the entire record, then they will set the JEDI_RECORD_FIELDREADWRITE bit in the Flags parameter and set this parameter to be the field number to read in. If the database driver does not support this operation, you can ignore this field (see earlier description of JEDI_RECORD_FIELDREADWRITE).

RETURN VALUE:

0 is returned if the record is read in successfully.

ENOENT is returned if the record does not exist.

Any other value shows the reason the operation failed using the values defined in errno.h.

JediWriteRecord: Write a record to a file.

This function is called when the application wants to write a record to the file.

SYNOPSIS:

```
int JediWriteRecord(FileDescriptor , Flags ,
                    RecordKey ,RecordKeyLen ,
                    BufferPtr , BufferLen,
                    FieldNumber)
JediFileDescriptor  * FileDescriptor;
int                   Flags ;
char                * RecordKey ;
int                   RecordKeyLen ;
char                * BufferPtr ;
int                   BufferLen ;
int                   FieldNumber ;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.

Flags. One or more of the following bits

- JEDI_RECORD_FIELDREADWRITE. If set, then the value at parameter 'FieldNumber' contains the field number (or attribute) to write, instead of the entire record. If the database drive doesn't support the writing of individual fields, then it will not set the JEDI_STATUS_FIELDWRITE bit during the JediOpen function and this will be ignored.

RecordKey. This points to an array of characters that define the record key for which to write the record as. The array is not a 0 terminated string.

RecordKeyLen. Shows the length of the record key as pointed to by the parameter RecordKey.

BufferPtr. This points to the actual data to write out, and is of length BufferLen bytes.

BufferLen. This shows the amount of data to be written, as pointed to by the BufferPtr parameter.

FieldNumber. If the caller wishes to write an individual field instead of the entire record, then they will set the JEDI_RECORD_FIELDREADWRITE bit in the Flags parameter and set this parameter to be the field number to write out. If the database driver does not support this operation, then this will be ignored.

RETURN VALUE:

The return value is 0 if the operation succeeded.

Any other value shows the reason the operation failed using the values defined in errno.h.

JediDelete: Delete a Record from a File.

This function is called when the application wants to delete a record from the file.

SYNOPSIS:

```
int JediDelete(FileDescriptor , Flags ,
                   RecordKey , RecordKeyLen )
JediFileDescriptor   * FileDescriptor;
int                    Flags ;
char                 * RecordKey ;
int                    RecordKeyLen ;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.

Flags. This parameter not used at present.

RecordKey. Points to a string describing the record key to delete Note the record key is not a 0 terminated string.

RecordKeyLen. This is the length of the record key as passed by RecordKey.

RETURN VALUE:

0 is returned if the record is deleted successfully.

ENOENT is returned if the record did not exist.

Any other value shows the reason the operation failed using the values defined in errno.h.

JediClearFile: Delete All Records from a File.

This function is called by an application to delete all the records from a file. No file locks are taken, so other applications could be writing to the file at the same time.

SYNOPSIS:

```
int JediClearFile(FileDescriptor)
    JediFileDescriptor  *FileDescriptor;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.

JediLock: Provide Record Locking Mechanism.

This function is called by the application to provide record locking support. It can also be called intrinsically via the jEDI code to support record locking functions. For example, if an application performs a request to READ a record with a lock, then the jEDI function will call the JediLock function directly, rather than the JediReadRecord function doing the call itself.

SYNOPSIS:

```
int JediLock(FileDescriptor , Flags ,
                 RecordKey , RecordKeyLen )
 JediFileDescriptor     * FileDescriptor;
int                       Flags ;
char                    * RecordKey ;
int                       RecordKeyLen ;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.
Flags. Can be one of the following bits to show what operation is to be performed by the LOCK function:
JEDI_RECORD_LOCKRECORD Take a record lock and wait.
JEDI_RECORD_LOCKRECORD_NOWAIT Take a record lock, but return if lock already taken (The function return code is JEDI_ERRNO_LOCK_TAKEN).
JEDI_RECORD_UNLOCKRECORD Remove a record lock.
RecordKey. This is a pointer to a character array, which shows what record key is to be locked or unlocked. Note that this array is not a 0 terminated string (See RecordKeyLen parameter).
A special case exists for when NULL is passed in this field. This means the calling application wants to release all the locks for all the records in the file held by this application.
RecordKeyLen. This is the length of data passed in the RecordKey parameter.

RETURN VALUE:

0 shows the locks were created/removed okay.
JEDI_ERRNO_LOCK_TAKEN shows the JEDI_RECORD_LOCKRECORD_NOWAIT bit was set, and the record/field was already locked by another process.
An EDEADLK show one of the lock bits was set, and a deadly embrace situation was detected and avoided.
Any other value shows the reason the operation failed using the values defined in errno.h.

JediIOCTL: Database driver control functions.

This function is called to provide general database driver operations. Like the ioctl() function in ‘C’ programs, there are some standard commands that most database drivers will be expected to perform, some that are optional, and some that are unique to the database driver.

SYNOPSIS:

```
int JediIOCTL(FileDescriptor , SubCommand ,
                  IoctlAddr , IoctlLen ,
                  IoctlReturnAddr , IoctlReturnLen)
JediFileDescriptor  * FileDescriptor;
int                   SubCommand  ;
void                * IoctlAddr ;
int                   IoctlLen ;
void                * IoctlReturnAddr ;
int                 * IoctlReturnLen;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.
SubCommand. This shows actually what sort of operation to perform. There are some common commands given by the JEDI_IOCTL_xxx definition, but the database driver can add support for other commands if required.
IoctlAddr. This gives the address of any optional buffer that may be necessary to provide extra specification depending on the SubCommand. Therefore its use is highly dependent upon SubCommand. The application may pass NULL here.
IoctlLen. This is the size of data that is pointed to by the IoctlAddr parameter.
IoctlReturnAddr. This is the address of a buffer or length IoctlReturnLen bytes where the IOCTL function can return any data depending upon the command to be executed (See SubCommand).
IoctlReturnLen. This is the address of an integer that shows the length of the return buffer pointed to by IoctlReturnAddr. Upon exit, the database driver should update this to show the amount of valid data that the JediIOCTL function has returned in IoctlReturnAddr.

RETURN VALUE:

The return value is highly dependent on the database driver. By convention though -1 is returned if the operation failed, 0 if the operation was successful.

JediSync: Synchronize the Data to Disk.

This function is called to flush any cache data to the disk. Not all database drivers support this - those that don’t simply return 0.

SYNOPSIS:

```
int JediSync(FileDescriptor)
    JediFileDescriptor  *FileDescriptor;
```

PARAMETERS:

FileDescriptor. This is the file descriptor that was returned when the file was originally opened.

RETURN VALUE:

The return value is 0 if the operation succeeded.
Any other value shows the reason the operation failed using the values defined in errno.h.