DB/C FS 101
Guide and Reference

January 2023

Table of Contents

Chapter 1. Installation and Operation

1.1 Introduction

DB/C FS is a two tier file and SQL server software product. The DB/C FS file server capability provides the traditional DB/C file access methods: sequential, random, indexed and associative indexed. The DB/C FS SQL server capability provides SQL access methods. Both file and SQL access methods work with traditional DB/C files.

The DB/C FS distribution consists of both server software and client software.

The server software consists of the file server program and an administrative tool.

The client software consists of SQL drivers and other software interfaces. The SQL interfaces are an ODBC driver and a JDBC driver. The ODBC driver is available for Microsoft Windows and for unixODBC on Linux. The JDBC driver works with Java applications that use the JDBC interface. The client software interfaces that provide access to the file server capability are a Java DB/C FS file access class library, C file access source code, and a .NET assembly. In addition, the current release of DB/C DX includes a file access interface to the DB/C FS file server.

1.2 Software

The client software distributed with all versions of DB/C FS consists of:

`setupodb32.exe`	the Windows 32 bit ODBC driver setup program
`setupodb64.exe`	the Windows 64 bit ODBC driver setup program
`setupodb`	the unixODBC driver setup program for Linux
`fsodbc.a`	the unixODBC driver for Linux
`fsodbcu.a`	the unixODBC driver DSN configuration support for Linux
`dbcfsjdbc.jar`	the JDBC driver program
`fs.jar`	the Java file interface package
`fs.dll`	the .NET class library assembly
`fsfileio.c`	the C file access interface source code file
`fsfileio.h`	the C file access interface header include file
`tcp.c`	an additional C interface source code file
`tcp.h`	an additional C interface header include file

The administrative programs are:

`dbcfsadm.exe`	the server administration program for Windows
`dbcfsadm`	the server administration program for Linux
`fscfg.zip`	the DB/C FS Configuration Files Editor for Eclipse

The server software and file utilities distributed with the Windows version of DB/C FS consists of:

`dbcfs.exe`	the file server program
`dbcfsrun.exe`	file server auxiliary program (do not run directly)

The server software and file utilities distributed with the Linux version of DB/C FS consists of:

`dbcfs`	the file server program
`dbcfsrun`	file server auxiliary program (do not run directly)

1.3 Installation of server and utilities software

To install the server software and administrator program for Windows, copy **`dbcfs.exe`**, **`dbcfsrun.exe`** and **`dbcfsadm.exe`** into a directory in the shell PATH.

To install the server software and administrator program for Linux, copy **`dbcfs`**, **`dbcfsrun`** and **`dbcfsadm`** into a directory that is included in the shell PATH.

1.4 Installation and setup of the Windows ODBC driver

To install the ODBC client software in Windows, at the command prompt run the DB/C FS ODBC setup program named **`setupodb.exe`**. This program steps you through the installation process. The setup program will copy several files into one or more Windows system directories. The setup program adds the DB/C FS driver as an available ODBC driver. If an old DB/C FS ODBC driver was installed previously, then you may want to uninstall it first using the “Add/Remove Programs” feature of Windows prior to the installation. Once the new DB/C FS driver is installed, you will not have an option to just uninstall older DB/C FS ODBC driver as both drivers must be uninstalled together. The DSN setup is the same as for other ODBC drivers.

1.5 Installation and setup of the unixODBC driver

To install the unixODBC client software in a Linux, make sure unixODBC is installed. Linux packages managers like yum and apt-get may be used to install and update unixODBC. Many or all of the command line programs described in this section must be run as a privileged user (e.g. root) which is typically done by using the **`sudo`** command as a prefix to the command line.

Change the current directory of the command line shell to the temporary distribution directory that contains the DB/C FS software. Run the **`setupodb`** program provided with DB/C FS from the command line like this:

```
./setupodb
```

This program checks to see unixODBC is installed and then asks for the name of an existing directory which is where it will copy the **`fsodbc.a`** and **`fsodbcu.a`** files to (the default is **`/usr/local/lib`** which may not be appropriate for your Linux distribution).

To configure the DB/C FS unixODBC driver, create the following file and name it **`fsdriver.tmp`**:

```
[dbcfs]
Description = FS ODBC Driver
Driver      = /usr/local/lib/fsodbc.a
Setup       = /usr/local/lib/fsodbcu.a
FileUsage   = 1
```

Be sure to use the correct fully-qualified path name where you copied the two files to.

Run the following command to install the unixODBC driver:

```
odbcinst -i -d -f fsdriver.tmp
```

The DB/C FS driver is installed. You can delete the **`fsdriver.tmp`** file.

Here is an example of how to create and install a DSN. Create the following file and name it **`mydsn.tmp`**:

```
[mydsn]
Description = Accounting System
Driver      = dbcfs
Database    = accounting
Server      = myserver.com
UID         =
Password    =
ServerPort  =
LocalPort   =
Encryption  = yes
```

Replace the values in this example with your own. The values corresond with the values described in the ODBC Considerations section of this document. Run the following command to install the DSN:

```
odbcinst -i -s -f mydsn.tmp
```

The DSN is installed. `mydsn.tmp` may be deleted.

1.6 Installation and setup of the JDBC driver

To install the JDBC client software in a Java environment, make the `dbcfsjdbc.jar` file available to the Java runtime.

1.7 Running DB/C FS server

The command line syntax to start the DB/C FS server is:

```
dbcfs config-file -fs=executable-file —g -x
```

where *config-file*, `-g`, `-x` and `-fs` are optional. If *config-file* is not specified, it defaults to `dbcfs.cfg`. If `-g` is specified in combination with the logging feature being specified in the configuration file, then the server will rename any existing log file prior to opening the specified log file in the configuration file. If `-x` is specified in combination with the logging feature being specified in the configuration file, then the server will suspend logging changes to the log file in the specified in configuration file. The `-fs` parameter is the alternate `dbcfsrun` name parameter which specifies a fully qualified alternate name of the `dbcfsrun` executable file.

1.8 Running DB/C FS server as a Linux background process

When executing DB/C FS server (`dbcfs`) as a background process in Linux, specify the `-y` option to cause `dbcfs` to continue to run even after the terminal connection used to start `dbcfs` is closed. This causes display of information that is normally sent to stdout to be redirected to `/dev/null`. Here is what the command line might look like:

```
dbcfs config-file -y &
```

1.9 Installing and running DB/C FS server as a Windows service

The Windows DB/C FS server may be installed as a service in Windows. These additional command line options provide for installation and removal of DB/C FS server as a service:

`-display=`*service-display-name* is used with the -service parameter and defaults to DB/C FS Service.

`-install` is the install parameter. This parameter causes DB/C FS to be installed as a Windows service in the stopped state.

`-password=`*password* is the login password parameter. It is optionally used with the **`-user`** parameter.

`-service=`*service-name* is useful if you have 2 licenses of FS and want to run both concurrently on the same system. The default value is **DbcfsService** and must be different for the additional instances of the DB/C FS server services. The service display name should also be changed using the **-display** option.

`-start` is the start parameter which can also be accomplished from the Services Control Panel or by restarting the computer. This option may be used in conjunction with the **-install** parameter or by itself to start a stopped service.

`-stop` is the stop parameter which can also be accomplished from the Services Control Panel and from the DB/C FS file server administration program.

`-uninstall` is the uninstall parameter which also performs an implicit **`-stop`** if necessary.

`-user=`*login-user-name* is used to assign a user name to the service. The *login-user-name* should be in the format of *domain-name*\*user-name*. If this parameter is not specified, then the **LocalSystem** account will be used. This can also be accomplished with the Services Control Panel prior to starting the service. A user name may need to be defined if required network drives or drive mappings are not available with the LocalSystem account. If the user name requires a password, the **`-password`** parameter must be specified.

`-verbose` causes various success or failure messages to be displayed.

The Services Control Panel can be used to start, stop and check the running status of the service. If an error occurs which causes **`dbcfs`** to stop execution, the error will usually be logged in the Application Log which can be viewed with the Event Viewer.

1.10 Considerations for using encryption

Encryption for communications between the FS server and client software uses industry standard SSL key managment. A key file containing a certificate and a private key for the certificate need to be generated and stored in a key file.

Here is an example of the commands to make the key file on Linux (substitute your own location and company name in the string following **`-subj`**):

```
openssl genrsa -out selfsigned.key 4096
openssl req -new -key selfsigned.key -out selfsigned.csr  -sha256 \
            -subj "/C=US/ST=Wyoming/L=SelfSigned/O=Portable Software/OU=Org/CN=localhost"
```

```
openssl x509 -req -days 9000 -in selfsigned.csr -signkey selfsigned.key -out selfsigned.crt
```

```
cat selfsigned.crt selfsigned.key > dbcserver.crt
```

Here is an example of the Power Shell commands to make the key file on Windows (substitute your own location and company name in the string following `-subj`):

```
openssl genrsa -out selfsigned.key 2048     // 4096 is an option
```

```
openssl req -new -key selfsigned.key -out selfsigned.csr -sha256
     -subj "/C=US/ST=Wyoming/L=SelfSigned/O=Portable Software/OU=Org/CN=localhost"
```

```
openssl x509 -req -days 9000 -in selfsigned.csr -signkey selfsigned.key -out selfsigned.crt
```

```
Get-Content .\selfsigned.crt, .\selfsigned.key | dbcserver.crt
```

The key file (`dbcserver.crt` in the examples) can have any name, but dbcserver.crt is the default name used by DB/C FS. By default, that file should be located in the current directory that is effective with the server starts. The file name and location can be modified with the **certificatefilename** runtime property.

1.11 The DB/C FS server adminstration program

The DB/C FS server administration program (**`dbcfsadm`**) provides for the display a list of the users that are connected, shut down the DB/C FS server, control logging, and remote execution of utility commands. The server administration program is run from the command line. Here is a list of command line options:

`dbcfsadm` *servername* **`-port=`***n* **`-noencrypt -p=`***password* **`-s -u -g -l -x -c="`***utility command line***`" -?`**

All command line parameters are optional.

servername specifies the host machine where the DB/C FS server is running. The default is the local machine.

-port overrides the default TCP/IP port number.

`-noencrypt` specifies that encryption is not used.

`-p` specifies the password for administrative operations if required.

`-s` causes the server task is stopped. The password parameter is necessary if the adminpassword property is specified in the server configuration file.

`-u` causes a list of users that are connected with the server to be displayed displayed. The password parameter is necessary if showpassword property is specified in the server configuration file. The **-u** and **-s** options are mutually exclusive.

`-g` causes the DB/C FS server task will to rename the log file. In addition, it will cause the DB/C FS server to log the modified records to the log file. This option is valid only if the logging feature is specified in the configuration file. The password parameter is required if the adminpassword property is specified in the configuration file.

`-l` causes the server to start logging changes to the log file. This option is similar to the -g option without the renaming. This option is valid only if the logging feature is specified in the configuration file. The password parameter is necessary only if the corresponding property is specified in the server configuration file.

`-x` causes the server to stop logging changes to the log file. This option is valid only if the logging feature is specified in the configuration file. The password parameter is necessary if the adminpassword property is specified in the server configuration file.

`-c` causes the server to execute the utility command using the specified command line arguments. The password parameter is required if the adminpassword property is specified in the server configuration file.

`-?` causes a help message will appear.

1.12 ODBC considerations

The DB/C FS ODBC client software supports 3 types of connections. The most common method is connecting with a Data Source Name (DSN) using either a User DSN or a System DSN. The other methods are to use a File DSN and to use a DSN-less connection.

To set up a User or System DSN, you can use the Windows ODBC Administrator to add a new DSN, or you can use the method specified in 1.4 to set up a DSN in LINUX. Here is the meaning of the parameters specified for a DSN:

`Data Source Name:`

The Data Source Name is the ODBC DSN name that is used by an application to connect to the associated driver. The DSN is also the name of the data dictionary file that is available to the DB/C FS server (see Data Dictionary File). The data dictionary file name may be case sensitive depending on the server. The name of the data dictionary file can be overridden using the Database Name entry. This field is required.

`Description:`

The Description is for your information only and is not used in the connection.

`Database Name:`

The Database Name specifies the name of the data dictionary file that is available to the DB/C FS server (see Data Dictionary File). If this field is not specified, then the value of Data Source Name is used.

`Default User ID:`

The Default User ID specifies the default user id name if the UID is not specified on the connection. If this field is not specified and the UID is not specified on the connection, then a username of **DEFAULTUSER** is used.

`Default Password:`

The Default Password specifies the default password that will be used if a password was not specified on the connection. If this field is not specified and the password is not specified on the connection, then a password of **PASSWORD** is used.

`Server Name:`

The Server Name is the DNS name of the computer or its IP address. This field is required.

`Server Port Number:`

The Server Port Number specifies the port number to use when connecting to the server. If this field is not specified, then port number 9584 is used for non-encrypted connections and port number 9585 is used for encrypted connections. This field must be specified if the **nport** or the **eport** configuration options are specified in the server configuration file.

`Local Port Number:`

The Local Port Number defaults to a dynamic value that is passed to the server. If the client has a firewall and needs to specify a specific port for the server to connect back to the client, then this option should be set to the appropriate port number. If the **sport** configuration option is specified in the server configuration file, then the Local Port Number may be set to 0 (zero). This will cause the secondary connection to be made in the direction of the server and opening a port on the client's firewall will not be required.

`Encryption:`

The Encryption check box determines if the connection between the client and server is encrypted.

To set up a File DSN, you can use the ODBC Administrator to add a new File DSN. Select the DB/C FS driver and a dialog will prompt you in the same manner as the above User or System DSN, but the Database field is a required field. Upon successful completion a file is created by the ODBC Administrator that contains two or more of the ODBC driver specific keywords in the `[ODBC]` section, like this:

`Database` `=` *data-dictionary-file*
`UID` `=` *user-name*
`Password` `=` *password*
`Server` `=` *server-address*
`ServerPort` `=` *port-number*
`LocalPort` `=` *port-number*
`Encryption` `=` `Yes` or `No`

A DSN-less connection can be created by specifying the File DSN keywords as above in combination with the Driver keyword. The minimum required keywords are Database and Server.

The File DSN keywords can also be used in conjunction with a User or System DSN connection to override the User or System DSN values.

Certain ODBC client software programs (e.g.Microsoft Access) do not provide a user name and password when connecting with the server. The user name defaults to the value of the Default User ID field previously described. If the Default User ID is not specified, the value **DEFAULTUSER** is used. The password defaults to the value of the Default Password field previously described. If the Default Password is not specified, the value **PASSWORD** is used. So for these applications, if you have not specified a Default User ID value and Default Password, you will need to have an entry in your password file like this:

`name` `=` `DEFAULTUSER`
`password` `=` `PASSWORD`
`access` `=` *access-code*

1.13 JDBC considerations

The DB/C FS JDBC protocol is `fs`. The JDBC driver class name is `com.dbcswc.fs.jdbc.Driver`.

The URL specification for the connection in one of these two formats:

`jdbc:fs:`*server***`/`***database*

`jdbc:fs:`*server***`:`***port***`/`***database*

One or more optional attribute values maybe appended to the end of the URL specification in the format of

`";`*attribute-name***=***attibute-value*`"`

where the following attributes are supported:

`localport=`*port-number*

`encryption=on` or **`off`**

For a description of these attributes, see the ODBC discussion of the same name.

1.14 Installation of the Eclipse configuration editor

The Eclipse Configuration Editor plug-in provides a way to create and modify the configuration, password, and database definition files for DB/C FS within the Eclipse Integrated Development Environment (IDE). Eclipse is an open source IDE that works on Window, Mac OS X, LINUX and other operating systems. When any of the DB/C FS configuration, password, or database definition files are modified and saved within the IDE, the output format is as described in the next chapter.

The first step to using the FS Configuration Editor and Eclipse is to install Eclipse. Eclipse may be downloaded from ww.eclipse.org. Install Eclipse according to the instructions for your operating system.

You can install the FS Configuration Editor after you have installed Eclipse. The FS Configuration Editor is an Eclipse Feature. It is installed by using the Eclipse Update Manager. The FS Configuration Editor is distributed in the **fscfg.zip** file. Copy the zip file into a directory of your choice. The Feature can then be installed using the standard Install Features menu items of Eclipse by specifying the **fscfg.zip** file as an Update Site and then completing the installation normally.

There are three editors in the Eclipse plug-in: the Configuration Editor, the Password Editor and the Database Definition Editor. The action of these editors is self-explanatory. The meanings on the pages and fields specified in these editors is described in the next chapter.

Chapter 2. Configuration Files

2.1 General syntax

TThe DB/C FS configuration file, the DB/C FS password file and the DB/C FS database definition files are XML text files. The syntax of these files allows XML Declaration and Document Type Definitions (DTD) at the beginning of the file, but these elements are ignored. Sections of the configuration and data dictionary files may be commented. Comments start with **`<!--`** and end with **`-->`**. When using DDL commands or the DB/C FS Configuration Editor plug-in for Eclipse, the comments, DTDs, and the XML Declaration will be lost when the file is saved to disk.

Here is a list of characters that must are illegal in XML and must be replaced by entity references:

`&lt;` is the less than character (**<**)
`&gt;` is the greater than character (**>**)
`&amp;` is the ampersand character (**&**)
`&apos;` is the apostrophe character (`'`)
`&quot;` is the quotation mark (`"`)

For example, to add the following column description to a DBD file: "Customer's zip code & address", the XML would look like this:

```
<d>Customer&apos;s zip code &amp; address</d>
```

2.2 Configuration file

The default DB/C FS configuration file name is **`dbcfs.cfg`**. This name may be overridden when the server is started or when the server is installed as a Windows service. If the default name is used or if no directory is specified in the override file name, then when the DB/C FS server is started, it searches for the configuration file in the current directory. It is recommended to specify the full path name of file names when installing DB/C FS as a Windows service.

The configuration file must contain a **`<dbcfscfg>`** root element which contains one or more of the following child elements. Unless otherwise noted, all elements are optional and may only be specified once.

`<licensekey>` *18-digit-key* **`</licensekey>`**
The license key element is required. The number of concurrent users allowed to connect with the DB/C FS server is encoded in the license key.

<passwordfile> *filename* **</passwordfile>**
The password file element defines the file containing user names and passwords. If not specified, the default name of this file is **`dbcfspwd.cfg`**.

<nport> *port-number* **</nport>**
The nport element defines the non-encrypted port number. It is used to specify the TCP/IP port number used by the server to perform non-encrypted communication with the clients. This element is typically required only when conflicting applications or firewall software require DB/C FS to use a different port. The default non-encrypted connection port number is 9584.

<eport> *port-number* **</eport>**
The eport element defines the encrypted port number. It is used to specify the TCP/IP port number used by the server to perform encrypted communications with clients. This element is typically required only when conflicting applications or firewall software require DB/C FS to use a different port. The default encrypted port is 9585.

<sport> *port-number* **</sport>**
The sport element defines the starting port number of a range of secondary port numbers. It is used to specify the TCP/IP port numbers used by the server to accept secondary connections from the clients. This will cause participating client connections to have both TCP/IP connections to be directed towards the server, instead of the default secondary connection being directed back towards the client. In order for the client to participate in the single direction connections, the client application must specify the local port parameter as 0 (zero). It is possible to have some of the clients using the sport feature and others not using the sport feature. The sport element is typically required if there is a problem with the secondary connection and a client's firewall. The range of port numbers is from sport to (sport + user license count – 1), regardless of the number of clients using the sport feature. Be sure the range does not conflict with the port numbers specified by nport or eport, other applications or firewall software.

<certificatefilename> *filename* **</certificatefilename>**
The certificate file name element defines the file that contains the private key and self-signed certificate for encrypting communications with the client. If not specified, the default name of this file is **`dbcserver.crt`**.

<dbdpath>
 <dir> *directory* **</dir>** (may be repeated, at least one required)
</dbdpath>
The database definition file (DBD file) lookup path parameter defines the directory or directories that are searched to find DBD files.

<volume> (may be repeated)
 <name> *volume* **</name>** (required)

`<dir>` *directory* **`</dir>`** (may be repeated, at least one required)

`</volume>`

The volume element specifies the directory or directories that are searched to open a file when the :*volume* specification is contained in a filename in a table definition in the DBD file. This element may be specified multiple times. It may also be overridden by an operand in the DBD file.

`<filepath>`

`<dir>` *directory* **`</dir>`** (may be repeated, at least one required)

`</filepath>`

The data and index file lookup path element specifies the directory or directories that are searched to open a file when there is no `:`*volume* specification or directory specification in a file name in the database definition. A file that is to be created is created in the first directory specification if the preppath element is not defined. This parameter may be overridden by an operand in the DBD file.

`<preppath>`

`<dir>` *directory* **`</dir>`** (may be repeated, at least one required)

`</preppath>`

The prepare data and index file path element specifies the directory where a file is created when there is no `:`*volume* specification or the file does not exist in the path specified by the filepath element. The default is the first directory specified by the filepath element.

`<workdir>` *directory* **`</workdir>`**

The work file directory element specifies the directory in which temporary work files are created.

`<namecase>` *upper-or-lower* **`</namecase>`**

The file name case element specifies the case translation for all file names. The *upper-or-lower* value is either **`upper`** or **`lower`**. If **`upper`** is specified, then the entire filename (prefix and suffix) is converted to upper case. If **`lower`** is specified, then the entire filename is converted to lower case. This feature is only used in conjunction with file access.

`<pathcase>` *upper-or-lower* **`</pathcase>`**

The path name case element specifies the case translation for the directory portion of all file names. The *upper-or-lower* value has the same values and semantics as for namecase.

`<extcase>` **`upper`** **`</extcase>`**

The filename extension case element specifies the case for file suffixes that are not specified in the file access open operation. If this operand is not specified, the default extension is lower -case. If it is specified, the default extension is upper case. This feature is only used in conjunction with file access.

`<casemap>` *translation-specification-or-filename* **`</casemap>`**

The aimdex file case translation table element specifies a semicolon delimited list of translation specifications or the file name of a 256 byte binary file that contains the case translation table. This parameter is typically only required with case insensitive aimdex files (non-distinct) using international characters. Each translation specification is in the form of either *nnn*`:`*nnn* or *nnn*`-`*nnn*`:`*nnn* where *nnn* is the decimal value of a character. The value before the colon is the 'translate from' character or the start of the 'translate from' character range and the character after the colon is the 'translate to' character or the first character in the 'translate to' range. Multiple translation specifications are separated using a semicolon and the default of converting a-z to A-Z would be specified as 97-122:65. If the specified value is not a valid translation list, then the operand is assumed to be a file name of a 256 byte file, where the byte offset value contains the value to translate to, where zero is the first byte. In the previous translation example, byte offset 97 would contain the value 65, byte offset 98 would contain the value 66, etc.

`<collatemap>` *translation-specification-or-filename* **`</collatemap>`**

The collating sequence element specifies a semicolon delimited list of translation specifications or the file name of a 256 byte binary file that contains the collating sequence table. Each translation specification is in the form of either *nnn*`:`*nnn* or *nnn*`-`*nnn*`:`*nnn* where *nnn* is the decimal value of a character. The value before the colon is the 'translate from' character or the start of the 'translate from' character range and the character after the colon is the 'translate to' character or the first character in the 'translate to' range. Multiple translation specifications are separated using a semicolon and the default ASCII table would be specified as 0-255:0. To have upper case characters have equal collating sequence as their lower case counterparts, both 97-122:65 and 65-90:97 would work though they are not the same as compared to the characters with the value 91-96. If the specified value is not a valid translation list, then the parameter is assumed to be a file name of a 256 byte file. Each byte in the file corresponds with a character in the character set. The first byte of the file corresponds with the character whose value is zero, the second byte corresponds with the character whose value is one, and so on. The value of each byte specifies the collating sequence of the character set. Thus the character which has a value zero in the corresponding byte in the file is the lowest character in the collating sequence, etc.

`<nodigitcompression/>`

The suppress digit compression characters element specifies that the characters between decimal value 128 and 248, inclusive, are not considered compression characters, but are considered to be part of the text character set that is used in each data record.

`<memalloc>` *nnnn* **`</memalloc>`**
The total work area memory allocation element specifies the number of kilobytes of memory that is allocated for the work area of the DB/C FS server. The default value is 256.

`<memresult>` *nnnn* **`</memresult>`**
The result size work area element specifies the number of kilobytes of memory from the work area that is used to contain a SELECT result set. If the result set will not fit into this area, it will be stored in a temporary file. If multiple SELECT statements will be active concurrently, then be sure that the memory allocated as a result of the memalloc element is large enough to contain all of the result sets.

`<adminpassword>` *password* **`</adminpassword>`**
This element specifies the password that must be specified by dbcfsadm to shut down the DB/C FS server.

`<showpassword>` *password* **`</showpassword>`**
This element specifies the password that must be specified by dbcfsadm to show the currently connected users.

`<compat>` *compatability* **`</compat>`**
The compatability element specifies file name conversion method. Valid values for compatability are **`dos`**, **`rms`**, **`rmsx`**, and **`rmsy`**. See the DB/C DX documentation for more information.

`<encryption>` *encryption* **`</encryption>`**
This element specifies the availability of encypted communications with the client. Valid values for encryption are **`on`**, **`off`**, or **`only`**. The **`on`** value specifies that encryption is the default mode of operation, but unencrypted communication will occur if the client requests it. **`off`** specifies that only unencrypted communication is available. **`only`** specifies that only encrypted communication is available. The default is **`on`**.

`<noexclusivesupport/>`
This element specifies that exclusive open support is not available. This element is ignored by the Windows version of DB/C FS.

`<excloffset>` *nnnn* **`</excloffset>`**
This element specifies the file offset that will be used to implement the exclusive open operation. This element is ignored by the Windows version of DB/C FS. The default value is operating system dependent, and should usually not be changed.

`<recoffset>` *nnnn* **`</recoffset>`**
This element specifies the file offset that will be used to implement record locking. The default value is operating system dependent, and should usually not be changed.

`<fileoffset>` *nnnn* **`</fileoffset>`**
This element specifies the file offset that will be used to implement file locking. The default value is operating system dependent, and should usually not be changed.

`<deletelock>` *lock-specification* **`</deletelock>`**
This element specifies the granularity of the locks that are performed during the SQL delete operation. Valid values for *lock-specification* are **`record`**, **`file`** and **`off`**. The default value is **`record`** which causes each record read from the text file to be locked and then released upon the record deletion or a failed condition comparison. In addition the text file will be locked if an index is used to find the record and will be locked again when deleting any keys from the index files. If the element is specified as file, then a file lock will be issued for the entire execution of the delete statement. The value **`file`** reduces system overhead as only one file lock is performed on the text file and additional file locks will not be required during index access or modification. But at the same time, using the value of file may increase the wait time of other applications to gain access to the file. The value **`off`** turns off any record locks, but file locks will be performed on the text file to access and modify the index similar to the value of **`record`**. The use of **`off`** is at the discretion of the user and the integrity of the records is at risk if multiple users are accessing records simultaneously.

`<updatelock>` *lock-specification* **`</updatelock>`**
This element specifies the granularity of the locks that are performed during a SQL update. The valid values for *lock-specification* are **`record`**, **`file`** and **`off`**. The default value is **`record`** which causes each record read from the text file to be locked and then released upon the record update or a failed condition comparison. In addition the text file will be locked if an index is used to find the record and will be locked again when updating any keys from the index files. If the element is specified as file, then a file lock will be issued for the entire execution of the update statement. The value **`file`** reduces system over head as only one file lock is performed on the text file and additional file locks will not be required during index access or modification. But at the same time, using the value of file may increase the wait time of other applications to gain access to the file. The value **`off`** turns off any record locks, but file locks will be performed on the text file to access and modify the index similar to the value of record. The use of **`off`** is at the discretion of the user and the integrity of the records is at risk if multiple users are accessing the records simultaneously.

The **`<logchanges>`** element of the configuration file is used to specify logging options. This element contains these child elements:

```
<logchanges>
    <allfiles/>   (either allfiles or one or more file required)
    <logfile> filename </logfile>   (required)
    <file> filename or pattern </file>    (pattern may contain * and/or ?)
    <lognames> off or on </lognames>
```

```
        <logopenclose> off or on </logopenclose>
        <logtimestamp> off or on </logtimestamp>
        <logusername> off or on </logusername>
    </logchanges>
```

The allfiles element specifies that changes to all files should be logged. If changes to only certain files are to be logged, then the logfile element should be used instead. One of either allfiles or logfile must be specified in the **`<logchanges>`** section. The format of the log file is specified in Appendix B.

The lognames element specifies if the logfile's file modification entries will contain the file name. Setting the value to **`on`** will cause the logfile to be marginally larger, but doesn't require tracking the file handle from the open to the associated file modification entries. The default is **`off`**.

The log open/close element specifies whether or not file open and close operations are logged. The default is **`on`**.

The log time stamp element specifies that all entries in the log file will have a 16 digit time stamp. The default is **`off`**. The format of the time stamp is:

yyyymmddhhmmsspp

where *yyyymmdd* is the date, *hhmmsspp* is the time. In some runtime environments, *pp* is hundredths of a second. In other environments, *pp* is always 00.

The log user name element specifies that each entry in the log file will have a user name associated with it. The default is **`off`**.

2.3 Name and password file

The default DB/C FS name and password file name is **dbcfspwd.cfg**. This name may be overridden in the configuration file. If the default name is used or if no directory is specified in the override file name, then when the server is started, it searches for the password file in the same manner that it searches for the configuration file.

The name and password file must contain a **`<dbcfspwd>`** root element with one or more child **`<user>`** elements.

Each **`<user>`** element consists of the name, password and access elements. The syntax is:

```
    <user>   (may be repeated, at least one required)
        <name> username </name>   (required)
        <password> password </password>   (required)
        <access> access code </access>   (may be repeated, at least one required)
    </user>
```

The name element defines the name of a user. The user name may be up to 20 characters long. It is not case sensitive.

The password element defines the password associated with the preceding user name. The password may be up to 20 characters long. It is not case sensitive.

The access codes element defines an access code that associated with a particular user name. Multiple access code elements may be declared for a single user. Access codes are 1 to 20 character identifiers. Access codes are not case sensitive.

2.4 Data dictionary file

The database name specified in the connection or initialization operation is converted to a database dictionary definition file name (DBD file) by appending a **`.dbd`** extension. Note that in Linux, the name of the DBD file is case sensitive, whereas in Windows it is not. In ODBC drivers, the database name may also be referred to as the data source name (DSN). If the dbdpath element was specified in the DB/C FS configuration file, the server searches for the DBD file in each of the directories specified by the dbdpath operand. If the dbdpath operand was not specified, then the server searches for the data dictionary file in the current directory. The dbdpath element should be specified if the server is run as a Windows NT service.

The DBD must contain a **`<dbcfsdbd>`** root element and one or more of the following child elements. There are two groups of elements in a DBD file. The first group of elements consists of access and general information operands. The second group of elements defines the SQL database.

The first group of elements consists of the following settings:

```
    <access> access code </access>   (required)
    <volume>   (may be repeated)
        <name> volume </name>   (required)
        <dir> directory </dir>   (may be repeated, at least one required)
    </volume>
    <filepath>
        <dir> directory </dir>   (may be repeated, at least one required)
    </filepath>
```

The access element defines the access code associated with this database. This element is required for both SQL and file access connections and is the only element required for file access connections. One of the access codes of a user attempting to use this data source must match this access code.

The volume element specifies the directory or directories that are searched to open a file when the `:`*volume* specification is contained in a filename in a table definition in the DBD file. This element may be specified multiple times. It overrides any setting in the configuration file that specifies the same volume.

The filepath element specifies the directory or directories that are searched to open a file when there is no `:`*volume* specification in a filename in the database definition. This parameter overrides the filepath element specified in the configuration file.

The second group of elements optionally defines one or more SQL tables. These elements are ignored for file access connections. Each table is defined within a **`<table>`** element which contains these child elements:

```
<table>
    <name> tablename </name>   (required)
    <description> description </description>
    <textfile> filename </textfile>   (required)
    <textfiletype> standard or data or crlf or text </textfiletype>
    <indexfile> filename </indexfile>
    <filter>
        <column> name </column>   (required)
        <value> value </value>   (required)
    </filter>
    <fixedlength> nnnnn </fixedlength>   (optional)
    <variablelength> nnnnn </variablelength>
    <compressed/>
    <readonly/>
<table>
```

The name element is required, and contains the table's name. The name may contain the question mark (?) character which will enable table templating. This feature requires the same number of question marks to be present in the corresponding textfile and indexfile elements. This is useful if the text file name and index file names contain something like a company number, where it is not feasible to create multiple table element entries. This allows using one table element against multiple data files. If the question marks are contained to just the file name portion, then the tables will be returned when using wild cards with table or column information functions such as ODBC's SQLTables and SQLColumns. If any of the question marks are contained within the file path or volume, then wild cards are not supported and the full table name must be specified on the query to return any information in regards to the two previous functions.

The description element is a comment field.

The textfile element specifies the filename of the text file of this table. The file extension must be specified.

The textfiletype element specifies the type of text file. Values for type are: **`standard`**, **`data`**, **`crlf`** and **`text`**. **`standard`** means the text file is a DB/C standard text file. **`data`** means the file is a portable text file (linefeed is the end of record character). **`crlf`** means the file is a portable text file (carriage return, linefeed are the end of record characters). **`text`** means the file is a non-portable, operating system dependent text file. This operand is optional for DB/C type files, but is necessary for **`data`**, **`crlf`**, and **`text`** type files that are empty or contain only deleted records. Specifying this parameter may reduce the time required to open the text file.

The indexfile element specifies the filename of an index of the textfile associated with this table. This file may be either an ISAM or an AIM index. The extension must be specified. The ordering of the index files has an effect on the search engine when it is trying to select which index to use. Generally, if it is trying to decide between two ISAM index files, which appear to be equal, it will select the first listed. Additionally, the search engine favors ISAM index files over AIM index files. This can be altered by listing the AIM index files before the ISAM index files. In this case, if all of the search conditions are with the equality operator, the AIM index keys cover all of the search conditions and the search engine cannot find a non-duplicate ISAM index file with an exact match, then the AIM index will be used. The only caution to using the AIM index before ISAM index feature is if none of the search columns satisfy the aimdexed read criteria of a X-type or F-type search. In this case, the AIM index search will fail and the search will revert to processing the file sequentially.

The filter element specifies a column filter for this table. Only those records that have the column specified by name that have the value specified by value are considered to be part of the table. This parameter also implies that the file is read only.

The fixedlength element specifies that all of the records in the file are the same length and are not compressed. If *nnnnn* is not specified, the file is read when it is opened and the length of the first record defines the record length. If *nnnnn* is specified, it defines the record length. In either case, an error will occur if any record is encountered that is a different length. Specifying *nnnnn* may reduce the time required to open the file.

The variablelength element specifies the maximum record length for files containing records of various lengths. This parameter prohibits the use of SQL update statement.

The compressed element means that the file is a DB/C type compressed file. This parameter prohibits the use of the SQL update statement.

The readonly element means that the file will be read only (i.e. The SQL INSERT, UPDATE, DELETE and SELECT FOR UPDATE statements will fail). The readonly element may be required for files that are stored on read only media.

Each table can contain one or more column, child (`<c>`) elements. Each column `<c>` element can have several child elements which describe the column. These elements are:

```
<c>
    <n> name </n>   (required)
    <t> data type </t>   (required)
    <p> record position </p>
    <d> description </d>
    <l> label </l>
    <nonnegative/>
</c>
```

The n element defines the column name. It is required. The name value must be 1 to 32 characters; the first character must be alphabetic and the following characters must be alphanumeric or the underscore (_) character.

The t element defines the data type of the column. It is required. The value of data type must be one of these:

```
characters ( n )
numeric ( n )
numeric ( n,m )
numeric ( n.m )
date ( date-specification )
time ( time-specification )
timestamp ( timestamp-specification )
```

`char` is synonymous with **`character`**. **`num`** is synonymous with **`numeric`**.

For the character data type, *n* is the number of characters in the character column.

For the numeric data type specified with `(` *n* `)`, the column is integral and *n* is the maximum number of digits allowed in the column. For the numeric data type specified with `(` *n*`,`*m* `)`, *n* is the maximum number of digits (including digits to the right of the decimal point) and *m* is the number of digits to the right of the decimal point. For example 1234 would be represented by **`numeric(4)`** or **`numeric(4,0)`** and **`1234567.89`** would be represented by **`numeric(9,2)`**. For the numeric data type specified with `(`*n*`.`*m*`)`, *n* is the number of digits to the left of the decimal point, and *m* is the number of digits to the right of the decimal point. For example **`12345.678`** would be represented by **`numeric(5.3)`**.

For the date data type, the *date-specification* may be of any combination of **`YYYY`**, **`YY`**, **`MM`**, **`DD`**, slash (**`/`**), hyphen (**`-`**), period (**`.`**), colon (**`:`**). For example, **`date(YYYYMMDD)`** and **`date(YY/MM/DD)`**.

For the time data type, the *time-specification* may be of any combination of **`HH`**, **`MM`**, **`SS`**, one or more **`F`**s (fraction), slash (**`/`**), hyphen (**`-`**), period (**`.`**), colon (**`:`**). For example **`time(HHMM)`** and **`time(HH:MM:SS.FFF)`**.

For the timestamp type, the *timestamp-specification* may be of any combination of **`YYYY`**, **`YY`**, **`MM`**, **`DD`**, **`HH`**, **`MM`**, **`SS`**,one or more **`F`**s, slash (**`/`**), hyphen (**`-`**), period (**`.`**), colon (**`:`**). The **`MM`** is assumed to represent months unless it's the first occurrence after HH; therefore minutes can not be specified before hours. For example **`timestamp(YYYYMMDDHHMMSS)`** and **`timestamp(YYYYMMDDHHMMSSFF)`**.

The position element (**`p`**) is optional. It specifies the starting position of the column in the record. This parameter is only necessary when a column is not contiguous with the previous column. The first position in a record is position 1.

The description element (**`d`**) is optional. It contains a description of the column.

The label element (**`l`**) is optional. It contains a label of the column that may be beneficial to the display of column headers in some applications. The label can be a maximum of 32 characters.

The nonnegative element is optional. It is useful for nonnegative numeric columns that make up an index key. This enables DB/C FS to use the index with that column, which may not be an option on certain comparison predicates against a numeric column containing negatives.

See Appendix A for a sample DBD file.

Chapter 3. SQL Syntax

3.1 Overview

DB/C FS supports a subset of ANSI/ISO Standard SQL. DB/C FS supports the four basic SQL data manipulation statements: SELECT, INSERT, DELETE and UPDATE. In addition, DB/C FS supports the following SQL data definition statements: ALTER, CREATE and DROP.

Cursors are supported. Positioned DELETE and UPDATE are supported. Fetch is supported for obtaining rows of a result set. The cursor support is accomplished outside of the syntax of the SELECT, DELETE and UPDATE statements. Rather it is accomplished with additional parameters and functions in the protocol used to communicate between the client and the server. Client software varies in its approach to cursors. Some client software adds syntax to the SQL statements to support cursors. Other client software implements cursor specification outside of the SQL statement syntax.

3.2 SQL data manipulation statements

The specific SQL data manipulation statements (DML) that DB/C FS supports are summarized in the following. Those elements enclosed in square brackets ([ ]) are optional. Elements separated by the vertical bar (**|**) mean that one or the other is allowed.

SELECT [**DISTINCT**] *select-list* **FROM** *table-list* [**WHERE** *search-condition*] [**ORDER BY** *sort-spec-list*]
[[**GROUP BY** *sort-spec-list*] [**HAVING** *having-search-condition*]] **|**
[[**FOR READ ONLY | FOR UPDATE** [**OF** *column-list*] [**NOWAIT**]]]

UPDATE *table-name* **SET** *column-set-value-list* [**WHERE** *search-condition*]

DELETE FROM *table-name* [**WHERE** *search-condition*]

INSERT [**INTO**] *table-name* **(** *column-list* **) VALUES (** *value-list* **)**

select-list is:
*** |** *column-list*
where ***** refers to all columns of all tables of this query.

column-list is a comma delimited list of one or more of these:
column-spec [[**AS**] *column-label*]
table-name **.***

table-label **.***
string-literal [[**AS**] *column-label*]
result-expression [[AS] *column-label*]

column-spec is:
column-name
*table-name***.***column-name*
*table-label***.***column-name*

*table-name***.*** and *table-label***.*** refer to all columns of the referenced table.

A *result-expression* is in algebraic form that contains constants, arithmetic operators, functions and columns. Functions consist of set functions, string functions, and CAST expressions.

Here is the list of set functions:
COUNT(*)
COUNT([**ALL | DISTINCT**] *qualified-column-name* **)**
AVG([**ALL | DISTINCT**] *qualified-column-name* **)**
MAX([**ALL | DISTINCT**] *qualified-column-name* **)**
MIN([**ALL | DISTINCT**] *qualified-column-name* **)**
SUM([**ALL | DISTINCT**] *qualified-column-name* **)**
In each of the set functions, **ALL** is the default. If **DISTINCT** is specified more than once in any expression in a SELECT statement, it must refer to the same column in all of the entries in this select statement that have **DISTINCT** specified.

The string value functions are:
UPPER(*value-expression* **)**
LOWER(*value-expression* **)**
value-expression **||** *value-expression*
SUBSTRING(*value-expression* **FROM** start [**FOR** length] **)**
TRIM ([**LEADING** | **TRAILING** | **BOTH**] [char-expression] **FROM** char-expression **)**
Note that **||** is the concatenate operator.

The **CAST** function is:
CAST(*value-expression* **AS** *datatype* **)**

value-expression is:

column-name
table-name **.** *column-name*
table-label **.** *column-name*
string-literal
string value function

table-list is:

table-spec [*join-operator table-spec* [**ON** *join-search*] . . .]

table-spec is:

table-name [[**AS**] *table-label*]

join-operator is one of:

,
[**INNER**] **JOIN**
LEFT OUTER JOIN

The comma operator and **INNER JOIN** are the same.

The **ON** search-condition clause is essentially the same as **WHERE**, except that it can only contain operators and columns from the two tables being joined.

search-condition is an expression in algebraic form that contains constants, operators, string functions and columns from the table or tables specified in the statement. The operators are:

comparative operators:	**= <> < > <= >= LIKE BETWEEN IN**
logical operators:	**AND OR NOT**
arithmetic operators:	**+ - * /**

column-set-value-list is one or more of the following comma separated values:

column-name **=** *value*

sort-spec-list is one or more of these comma delimited values:

column-spec
column-label
result-set-column-number

having-search-condition is a *search-condition* which may optionally contain expressions using set functions.

column-list is a comma delimited list of *column-names* and *column-labels*.

column-name refers to a column defined in the DBD file. If a *column-label* is associated with a *column-name* by an **`AS`** clause, it refers to that column defined in the DBD file.

table-name refers to a table defined in the DBD file.

In a SELECT statement, a column name may be unqualified or qualified with a *table-name* or a *table-label* that refers to a *table-name*, which thus refers to a table defined in the DBD file.

value-list is a list of string and numeric values.

Sub-queries are not supported.

The use of **`<>`**, **`NOT LIKE`**, **`NOT BETWEEN`**, **`NOT IN`** and dissimilar **`OR`** conditions are not recommended on large files as indexes will not be utilized and the entire text file will be read sequentially.

The tables in a **`JOIN`** are processed from left to right. With this in mind, the tables should be ordered so that the left table(s) can provide data which can be used with the indexes of the right table(s). Doing this will enhance performance.

3.3 SQL data definition statements

The specific SQL data definition statements (DDL) that DB/C FS supports are summarized with the following. Those elements enclosed in brackets ([ ]) are optional. Elements separated by the vertical bar (**`|`**) mean that one or the other is allowed. Note that DDL must be used carefully. DB/C FS server processes do not periodically check the DBD file for changes. Therefore, DDL commands should only be used on tables that are not currently being used by other connections.

> **`CREATE TABLE`** *table-name* **`(`** *column-definition* **`,`** . . . **`)`**

column-definition is:
> *column-name datatype* [**`(`** *length* **`)`**]

Example: **`CREATE TABLE TEST (ITEM CHAR(6), VEND CHAR(6), LIST NUMERIC(9,3), QYTO NUMERIC(6))`**

table-name is the unique name for the table. The table name will be stored in uppercase in the DBD file, regardless of what case was used in the CREATE statement.

CREATE [**UNIQUE** | **ASSOCIATIVE**] **INDEX** *index-name* **ON** *table-name* **(** *column-name* **, . . .)**

Example: `CREATE UNIQUE INDEX TEST1 ON TEST (VEND, ITEM)`

index-name specifies the name of the index. If no extension is specified, the extension `.isi` or `.aim` is appended accordingly. If the index file does not yet exist it will be created. If the column (key) specifications do not match the existing index file, then the index file is replaced with a new index file. New index files are created in the preppath directory specified in the configuration file. If preppath is not defined, then the first directory in filepath is used.

The **UNIQUE** option prevents the index file from allowing duplicate values. Insertion of non-unique value into the table with a **UNIQUE** index will result in an error. If instead, **ASSOCIATIVE** is specified, then an **AIM** file is created.

The column length parameter allows an index to be created on the first length characters of a CHAR column.

DROP TABLE [**IF EXISTS**] *table-name*

Example: `DROP TABLE TEST`

The **IF EXISTS** option can be added to prevent an error from occurring when attempting to drop a table that might not exist.

table-name is not case sensitive. When a table is dropped, it is removed from the DBD file, and all associated index files and text files are removed as well.

DROP [**ASSOCIATIVE**] **INDEX** *table-name* **.** *index-name*

Example: `DROP INDEX TEST.TEST1`

The *index-name* index will be deleted from disk. In addition, the index will be removed from the table tablename in the DBD. The `.aim` or `.isi` extension should be omitted on the index name.

ALTER TABLE *table-name alter-specification*

alter-specification is:

ADD [**COLUMN**] (column-name data-type [(length)] , . . .)
| **ADD** [**UNIQUE** | **ASSOCIATIVE**] **INDEX** index-name (column-name [(length)] , . . .)
| **DROP** [**COLUMN**] column-name
| **DROP** [**ASSOCIATIVE**] **INDEX** index-name
| **RENAME** [**TO**] new-table-name

Examples:

```
ALTER TABLE CUSTOMER ADD COLUMN LASTNAME CHAR(6)
ALTER TABLE CUSTOMER DROP COLUMN FAXNO
ALTER TABLE CUSTOMER DROP ASSOCIATIVE INDEX LCUSTOMER
ALTER TABLE CUSTOMER RENAME TO NEWCUST
```

For `DROP INDEX`, *index-name* should not have an extension. `DROP COLUMN` will fail if an index exists on the column. In addition, `DROP COLUMN` will not reformat the data file.

Chapter 4. Java Client Programming Interface

4.1 Overview

The Java client programming application programming interface provides Java classes that allow access to DB/C FS data files using traditional sequential, random, indexed and AIM indexed access methods. These classes are provided in the file named **`fs.jar`**. The classes contained in this file are all part of the **`com.dbcswc.fs`** package. The classes are: **`Connection`**, **`File`**, **`Util`**, **`ConnectionException`** and **`FileException`**.

4.2 Connection

The **`Connection`** class implements a connection to the DB/C FS server. An instance of this class is required by the **`File`** class for most of its functionality.

```
public static String getgreeting(String server) throws ConnectionException
```

```
public static String getgreeting(String server, int port, boolean encrypt, String authfile)
     throws ConnectionException
```

If the DB/C FS server specified by the **`server`** parameter is available, the **`getgreeting`** method returns a String that provides the version number of the DB/C FS server. The **`server`** parameter is the domain name or TCP/IP address of the computer running the DB/C FS server. The **`port`** parameter is the port to connect to on the server. Specify a value of -1 to use the default port number. If **`encrypt`** is true and JSSE is not available, a **`ConnectionException`** is thrown. The **`authfile`** parameter should be set to null. If the server is not available, a **`ConnectionException`** is thrown.

```
public Connection(String server, int serverport, int localport, String database,
     String user, String password, boolean encrypt, String authfile)
     throws ConnectionException
```

```
public Connection(String server, String database, String user, String password,
     boolean encrypt, String authfile) throws ConnectionException
```

```
public Connection(String server, int serverport, int localport, String database,
     String user, String password) throws ConnectionException
```

```
public Connection(String server, String database, String user, String password)
     throws ConnectionException
```

The **`Connection`** constructor creates a connection between this Java virtual machine and the DB/C FS server. The **`server`** parameter is either the domain name or the TCP/IP address of the computer running the DB/C FS server. The **`serverport`** parameter specifies the TCP/IP port number of the DB/C FS Server. If this value is –1, or is not specified, the default value is used. The **`localport`** parameter specifies the local TCP/IP port number that will be used by the server to connect back to complete the connection. If this value is -1 or is not specified, a dynamically created port number will be used. If this value is zero, then this signals the server that the connection should use the **`sport`** feature of the server. The database parameter is the name of the database to be accessed (i.e. the name of the DBD file, without the **`.dbd`** extension). The user and password parameters are the user id and password required for login to the DB/C FS server. If **`encrypt`** is true and JSSE is not available, a **`ConnectionException`** is thrown. The **`authfile`** parameter should be set to null. If the connection to the server cannot be established, a **`ConnectionException`** is thrown.

```
public void disconnect()
```

The **`disconnect`** method destroys this connection with the DB/C FS server.

```
public String getserver()
```

The **`getserver`** method returns the name of the server associated with this **`Connection`** instance.

```
public String getdatabase()
```

The **`getdatabase`** method returns the name of the database associated with this **`Connection`** instance.

```
public String getuser()
```

The **`getuser`** method returns the name of the user associated with this **`Connection`** instance.

4.3 File

The **`File`** class implements sequential, random, indexed and AIM indexed data access to a DB/C FS server. An instance of the **`Connection`** class is required by **`File`** for most of its functionality. The following public fields are used by the **`setoptions`** and **`getoptions`** methods.

```
public static final int VARIABLE
public static final int COMPRESSED
public static final int TEXT
public static final int DATA
public static final int CRLF
public static final int BINARY
public static final int READONLY
public static final int EXCLUSIVE
public static final int DUP
public static final int NODUP
public static final int CASESENSITIVE
public static final int LOCKAUTO
public static final int LOCKSINGLE
public static final int LOCKNOWAIT
public static final int CREATEFILE
public static final int CREATEONLY
public static final int FILEOPEN
public static final int IFILEOPEN
public static final int AFILEOPEN
```

Each of the methods in `File` whose names start with `read` returns a record in the record buffer. The integer value returned from these methods is the number of characters in the record that was just read, or -1 if the record was not found or the read position was at the end of file, or -2 for those methods that end with `lock` when the lock could not be obtained and the `LOCKNOWAIT` option was specified.

```
public File(Connection fs)
```

The `fs` parameter of the `File` constructor is a `Connection` to a DB/C FS server.

```
public void setdatafile(String datafilename)
```

The `datafilename` parameter of `setdatafile` specifies the data file name to be used during the open method. This method must be called before the open method is called. It is optional if the setindexfile method is called, otherwise it is required for open to be successful.

```
public void setindexfile(String indexfilename)
```

The `indexfilename` parameter of `setindexfile` specifies the index file name to be used during the open method. This method must be called before the open method is called. It is required for indexed file open operations.

```
public String getfilename()
```

The value returned is the data file name associated with this `File`.

```
public void setoptions(int options)
```

This method specifies the options for the following open operation. The value of the `options` parameter is the OR'd together value of the public fields described above. Exactly one of the `FILEOPEN`, `IFILEOPEN` and `AFILEOPEN` must be specified. All other values are optional. Certain combinations of values are ambiguous and should not be specified (e.g. T**`EXT`** and **`BINARY`**). The result is undefined for these ambiguous situations.

`public int getoptions()`
The value returned by **`getoptions`** is the OR'd together value of the public fields described above.

`public void setrecsize(int recsize)`
The **`recsize`** operand of the **`setrecsize`** method specifies the record size to be used during the following open method. If the **`VARIABLE`** option is specified, the **`recsize`** operand indicates the maximum possible size of a record. This method is required.

`public int getrecsize()`
This method returns the record size associated with the file . If the file contains variable length records, then the value returned is the maximum record size.

`public void setkeysize(int keysize)`
The **`keysize`** parameter of this method specifies the key size for the following open statement. This is required only when an indexed file is being created. If it is specified when an existing index is being opened and the value does not match the key size of the index, a **`FileException`** will be thrown by the open method.

`public void setkeyinfo(String keyinfo)`
The **`keyinfo`** parameter of this method specifies the AIM key information for the following open statement. This is required only when an AIM indexed file is being created. The value in **`keyinfo`** is a comma delimited list of field specifications. A field specification is either a positive integer or is two positive integers separated by a hyphen.

`public void setmatchchar(char matchchar)`
The **`matchchar`** parameter specifies the match character for AIM index operations.

`public void open() throws FileException`
The **`open`** method creates or opens the file with filename and other parameters as specified in the preceeding methods. If the file cannot be created or opened, a **`FileException`** is thrown.

`public void close() throws FileException`
The **`close`** method closes this **`File`** instance. A **`File`** instance may be opened and closed multiple times. If the instance is not in the open state, then a **`FileException`** is thrown.

`public void closedelete() throws FileException`

The **`closedelete`** method closes this **`File`** instance and deletes the associated data and index files. A **`File`** instance may be opened and closed multiple times. If the instance is not in the open state, then a **`FileException`** is thrown.

`public boolean isopen()`

The value returned by the isopen **`method`** is true if this **`File`** instance is in the open state, and is false otherwise.

`public void setrecordbuffer(char[] newrecbuf)`

The **`setrecordbuffer`** method specifies the character array that will be used to hold the record that is read or written by various of the following methods. This method is required.

`public long fposit() throws FileException`

The value returned by the **`fposit`** method is the value called the current file position. The current file position is the offset (in bytes) of the last record read from or written to the file, or is the value specified by the **`reposit`** method if it was called after the most recent read or write. The initial value returned by the fposit method immediately after a file is opened is zero. If the instance is not in the open state, then a **`FileException`** is thrown.

`public int reposit(long pos) throws FileException`

The **`reposit`** method sets the value of the current file position to the value specified by the **`pos`** parameter. The value returned indicates what the position is. The return value 0 means the current file position was set before end of file. The return value 1 means the current file position was set at end of file. The return value 2 means the current file position was not set because the value specified was past end of file.

`public int setposateof() throws FileException`

The **`setposateof`** method sets the current file position to the end of file.

`public int readrandom(int recnum) throws FileException`

The **`readrandom`** method reads a record into the record buffer. The record is read using the random access method. Random access records must be fixed length (**`VARIABLE`** cannot be specified as an open option). The record read is specified by the **`recnum`** parameter. The first record in the file is record number zero.

`public int readrandomlock(int recnum) throws FileException`

The **`readrandomlock`** method operates the same as the **`readrandom`** method, except that the record is locked.

`public int readnext() throws FileException`

The **`readnext`** method reads a record into the record buffer. The record is read using the sequential access method to retrieve the record that is after the record most recently read.

`public int readnextlock() throws FileException`

The **`readnextlock`** method operates the same as the readnext method, except that the record is locked.

`public int readprev() throws FileException`

The **`readprev`** method reads a record into the record buffer. The record is read using the sequential access method to retrieve the record that is before the record most recently read.

`public int readprevlock() throws FileException`

The **`readprevlock`** method operates the same as the **`readprev`** method, except that the record is locked.

`public int readsame() throws FileException`

The **`readsame`** method reads a record into the record buffer. The record that is read is the record at the current file position.

`public int readkey(String key) throws FileException`

The **`readkey`** method reads a record into the record buffer. The record is read using the indexed sequential access method. The **`key`** parameter specifies the key used to lookup the record.

`public int readkeylock(String key) throws FileException`

The **`readkeylock`** method operates the same as the **`readkey`** method, except that the record is locked.

`public int readkeynext() throws FileException`

The **`readkeynext`** method reads a record into the record buffer. If this file is accessed by the index sequential access method, then the next key sequential record is read. If this file is accessed by the AIM index access method, then the next record in the result set is read.

`public int readkeynextlock() throws FileException`

The **`readkeynextlock`** method operates the same as the **`readkeynext`** method, except that the record is locked.

`public int readkeyprev() throws FileException`

The **`readkeyprev`** method reads a record into the record buffer. If this file is accessed by the index sequential access method, then the previous key sequential record is read. If this file is accessed by the AIM index access method, then the previous record in the result set is read.

`public int readkeyprevlock() throws FileException`

The **`readkeyprevlock`** method operates the same as the **`readkeyprev`** method, except that the record is locked.

`public int readaim(String keyinfo) throws FileException`

This method reads the first record of a result set of records into the record buffer. The result set is created using the AIM index access method. The **`keyinfo`** parameter specifies the lookup criteria. The lookup criteria is a String of 4 or more characters. The first two characters are digits (blank filled in the first position) that specify the column for the search. The third character is **`F`** or **`X`** or **`L`** or **`R`** for floating, exact, left or right match, respectively. The remaining characters are the match characters. The result set consists of those records that match the lookup criteria.

`public int readaimlock(String keyinfo) throws FileException`
This method operates the same as the **`readaim`** method, except that the record is locked.

`public int readaim(String[] keyinfoarray) throws FileException`
This method reads the first record of a result set of records into the record buffer. The result set is created using the AIM index access method. The **`keyinfoarray`** parameter array specifies the multiple lookup criteria. The syntax of each of the lookup criteria is as specified above. The result set consists of those records that match all of the lookup criteria.

`public int readaimlock(String[] keyinfoarray) throws FileException`
This method operates the same as the **`readaim`** method, except that the record is locked.

`public void writerandom(int reclen, int recnum) throws FileException`
The **`writerandom`** method writes a record from the record buffer using the random access method. The length of the record written is specified by the **`reclen`** parameter. The record is written into the record number specified by the **`recnum`** parameter.

`public void writenext(int reclen) throws FileException`
The **`writenext`** method writes a record from the record buffer using the sequential access method. The length of the record written is specified by the **`reclen`** parameter. The record is written after the most recently written record.

`public void writeateof(int reclen) throws FileException`
The **`writeateof`** method writes a record from the record buffer using the sequential access method. The length of the record written is specified by the **`reclen`** parameter. The record is written at the end of the file.

`public void writekey(int reclen, String key) throws FileException`
The **`writekey`** method writes a record from the record buffer using the sequential access method and the indexed access method. The length of the record written is specified by the **`reclen`** parameter. The record is written at the end of the file. The value of the key parameter and the position of the record just written is added to the index file.

`public void writeaim(int reclen) throws FileException`
The **`writeaim`** method writes a record from the record buffer using the sequential access method and the AIM indexed access method. The length of the record written is specified by the reclen parameter. The record is written at the end of the file. The AIM index is updated to reflect the values of the AIM keys that were contained in the record just written.

`public void insertkey(String key) throws FileException`

This method inserts a key and the current file position into the index file.

`public void insertkeys(int reclen) throws FileException`

This method causes the AIM index is updated to reflect the values of the AIM keys that are contained in the record in the record buffer.

`public void update(int reclen) throws FileException`

This method updates a record from the record buffer. The length of the record written is specified by the reclen parameter. The record is written over the record at the current file position.

`public void delete() throws FileException`

This method deletes the record that was most recently read or written.

`public void delete(String key) throws FileException`

This method deletes a key from the index file and the record that it is associated with from the data file. The **`key`** parameter specifies the key that is deleted.

`public void deletekey(String key) throws FileException`

This method deletes the key that is specified by the **`key`** parameter from the index file.

`public void unlock(long pos) throws FileException`

This method unlocks the record at the file position specified by the **`pos`** parameter.

`public void unlock() throws FileException`

This method unlocks records locked by this instance of **`File`**.

`public void weof(long pos) throws FileException`

This method writes an end of file marker at the position specified by the **`pos`** parameter.

`public int compare(File file) throws FileException`

The **`compare`** method compares this **`File`** instance with the instance specified by the file parameter. The integer value that is returned is zero if the files are the same. The value will be less than zero if the files compare such that this **`File`** instance is less than the File instance of the file parameter. The value will be positive otherwise. The concept of less than is not defined, but will always be the same for comparison of the same files. This method is used for ordering filelock calls to avoid deadlock.

`public void filelock() throws FileException`

The **`filelock`** method locks this data file. If the data file is already locked, this method will wait for the data file to be unlocked by the other program.

`public void fileunlock() throws FileException`

The **`fileunlock`** method unlocks this data file.

Chapter 5. .NET Client Programming Interface

5.1 Overview

The .NET client programming application programming interface provides a .NET class library that allows access to DB/C FS data files using traditional sequential, random, indexed and AIM indexed access methods. These classes are provided in the assembly named **`fs.dll`**. The classes contained in this assembly are all part of the **`dbcfs`** namespace. The classes are: **`Connection`**, **`File`**, **`Util`**, **`ConnectionException`** and **`FileException`**. Encryption is not available.

5.2 Connection

The **`Connection`** class implements a connection to the DB/C FS server. An instance of this class is required by the **`File`** class for most of its functionality.

```
public static string getgreeting(string server)
public static string getgreeting(string server, int port, bool encrypt, string authfile)
```

If the DB/C FS server specified by the **`server`** parameter is available, then this method returns a value that provides the version number of the DB/C FS server. The server parameter is the domain name or the TCP/IP address of the computer running the DB/C FS server. The **`encrypt`** parameter is ignored. The **`authfile`** parameter should be set to null. If the server is not available, a **`ConnectionException`** is thrown.

```
public Connection(string server, int serverport, int localport, string database,
     string user, string password, bool encrypt, string authfile)
public Connection(string server, string database, string user, string password,
     bool encrypt, string authfile)
public Connection(string server, int serverport, int localport, string database,
     string user, string password)
public Connection(string server, string database, string user, string password)
```

The **`Connection`** constructor creates a connection between the client and the DB/C FS server. The **`server`** parameter is either the domain name or the TCP/IP address of the computer running the DB/C FS server. **`serverport`** specifies the TCP/IP port number of the DB/C FS Server. If this value is zero or is not specified, the default value is used. **`localport`** specifies the local TCP/IP port number that will be used by the server to connect back to complete the connection. If this value is –1 or is not specified, a dynamically created port number will be used. If this value is zero, then this signals the server that the connection should use the **`sport`** feature of the server. The database parameter is the name of the database to be accessed, which is the name of the DBD file with out the **`.dbd`** extension. The user and password parameters are the user id and password required for login to the server. authfile is ignored. If the connection to the server cannot be created, a **`ConnectionException`** is thrown.

`public void disconnect()`
This method destroys this connection with the DB/C FS server.

`public string getserver()`
This method returns the name of the server associated with this **`Connection`** instance.

`public string getdatabase()`
This method returns the name of the database associated with this **`Connection`** instance.

`public string getuser()`
This method returns the name of the user associated with this **`Connection`** instance.

5.3 File

The **`File`** class implements sequential, random, indexed and AIM indexed data access to a DB/C FS server. An instance of the **`Connection`** class is required by the **`File`** class for most of its functionality. The following fields are used by the **`setoptions`** and **`getoptions`** methods.

```
public static const int VARIABLE
public static const int COMPRESSED
public static const int TEXT
public static const int DATA
public static const int CRLF
public static const int EBCDIC
public static const int BINARY
public static const int READONLY
public static const int EXCLUSIVE
public static const int DUP
public static const int NODUP
public static const int CASESENSITIVE
public static const int LOCKAUTO
public static const int LOCKSINGLE
public static const int LOCKNOWAIT
public static const int CREATEFILE
public static const int CREATEONLY
public static const int FILEOPEN
public static const int IFILEOPEN
public static const int AFILEOPEN
```

Each of the methods in the **`File`** class whose names that start **`read`** return a record in the record buffer. The integer value returned from these methods is the number of characters in the record that was just read, or -1 if the record was not found or the read position was at the end of file, or -2 for those methods that end with **`lock`** when the lock could not be obtained and the **`LOCKNOWAIT`** option was specified.

```
public File(dbcfs.Connection fs)
```

The **`fs`** parameter of the **`File`** constructor is a **`Connection`** to a DB/C FS server.

```
public void setdatafile(string datafilename)
```

The **`datafilename`** parameter of thia method specifies the data file name to be used during the **`open`** method. This method must be called before the **`open`** method is called. It is optional if the **`setindexfile`** method is called, otherwise it is required for open to be successful.

```
public void setindexfile(string indexfilename)
```

The **`indexfilename`** parameter of this method specifies the index file name to be used during the **`open`** method. This method must be called before the **`open`** method is called. It is required for indexed file open operations.

```
public string getfilename()
```

The value returned is the data file name associated with this **`File`** instance.

```
public void setoptions(int options)
```

This method specifies the options for the following open operation. The value of the **`options`** parameter is the OR'd together value of the fields described above. Exactly one of the **`FILEOPEN`**, **`IFILEOPEN`** and **`AFILEOPEN`** must be specified. All other values are optional. Certain combinations of values are ambiguous and should not be specified (e.g. **`TEXT`** and **`BINARY`**). The result is undefined for these ambiguous situations.

```
public int getoptions()
```

The value returned by this method is the OR'd together value of the fields described above.

```
public void setrecsize(int recsize)
```

The **`recsize`** parameter of this method specifies the record size to be used during the following open operation. If the **`VARIABLE`** option is specified, the **`recsize`** operand indicates the maximum possible size of a record. This method is required.

`public int getrecsize()`

This method returns the record size associated with the file. If the file contains variable length records, then the value returned is the maximum record size.

`public setkeysize(int keysize)`

The **`keysize`** parameter of this method specifies the key size for the following open operation. This is required only when an indexed file is being created. If it is specified when an existing index is being opened and the value does not match the key size of the index, a **`FileException`** will be thrown by the **`open`** method.

`public void setkeyinfo(string keyinfo)`

The **`keyinfo`** parameter of this method specifies the AIM key information for the following open operation. This is required only when an AIM indexed file is being created. The value in **`keyinfo`** is a comma delimited list of field specifications. A field specification is either a positive integer or is two positive integers separated by a hyphen.

`public void setmatchchar(char matchchar)`

The **`matchchar`** parameter specifies the match character for AIM index operations.

`public void open()`

The **`open`** method creates or opens the file with filename and other parameters as specified in the preceeding methods. If the file cannot be created or opened, a **`FileException`** is thrown.

`public void close()`

The **`close`** method closes this **`File`** instance. A **`File`** instance may be opened and closed multiple times. If the instance is not in the open state, then a **`FileException`** is thrown.

`public void closedelete()`

The **`closedelete`** method closes this **`File`** instance and deletes the associated data and index files. A **`File`** instance may be opened and closed multiple times. If the instance is not in the open state, then a **`FileException`** is thrown.

`public bool isopen()`

The value returned by this method is true if this **`File`** instance is in the open state, and false otherwise.

`public void setrecordbuffer(char[] newrecbuf)`

This method specifies the character array that will be used to hold the record that is read or written by various of the following methods. This method is required.

`public long fposit()`

The value returned by this method is the current file position. The current file position is the offset (in bytes) of the last record read from or written to the file, or is the value specified by the reposit method if it was called after the most recent read or write. The initial value returned by the fposit method immediately after a file is opened is zero. If the instance is not in the open state, then a **`FileException`** is thrown.

`public int reposit(long pos)`
This method sets the value of the current file position to the value specified by the **`pos`** parameter. The value returned indicates what the position is. The return value 0 means the current file position was set before end of file. The return value 1 means the current file position was set at end of file. The return value 2 means the current file position was not set because the value specified was past the end of file.

`public int setposateof()`
This method sets the current file position to the end of file position.

`public int readrandom(int recnum)`
Thismethod reads a record into the record buffer. The record is read using the random access method. Random access records must be fixed length (**`VARIABLE`** cannot be specified as an open option). The record read is specified by the **`recnum`** parameter. The first record in the file is record number zero.

`public int readrandomlock(int recnum)`
This method operates the same as the readrandom method, except that the record is locked.

`public int readnext()`
This method reads a record into the record buffer. The record is read using the sequential access method to retrieve the record that is after the record most recently read.

`public int readnextlock()`
This method operates the same as the **`readnext`** method, except that the record is locked.

`public int readprev()`
This method reads a record into the record buffer. The record is read using the sequential access method to retrieve the record that is before the record most recently read.

`public int readprevlock()`
This method operates the same as the **`readprev`** method, except that the record is locked.

`public int readsame()`
This method reads a record into the record buffer. The record that is read is the record at the current file position.

`public int readkey(string key)`

This method reads a record into the record buffer. The record is read using the indexed sequential access method. The **`key`** parameter specifies the key used to lookup the record.

`public int readkeylock(string key)`

This method operates the same as the **`readkey`** method, except that the record is locked.

`public int readkeynext()`

This method reads a record into the record buffer. If this file is accessed by the index sequential access method, then the next key sequential record is read. If this file is accessed by the AIM index access method, then the next record in the result set is read.

`public int readkeynextlock()`

This method operates the same as the **`readkeynext`** method, except that the record is locked.

`public int readkeyprev()`

This method reads a record into the record buffer. If this file is accessed by the index sequential access method, then the previous key sequential record is read. If this file is accessed by the AIM index access method, then the previous record in the result set is read.

`public int readkeyprevlock()`

This method operates the same as the **`readkeyprev`** method, except that the record is locked.

`public int readaim(string keyinfo)`

This method reads the first record of a result set of records into the record buffer. The result set is created using the AIM index access method. The **`keyinfo`** parameter specifies the lookup criteria. The lookup criteria is a string of 4 or more characters. The first two characters are digits (blank filled in the first position) that specify the column for the search. The third character is **`F`** or **`X`** or **`L`** or **`R`** for floating, exact, left or right match, respectively. The remaining characters are the match characters. The result set consists of those records that match the lookup criteria.

`public int readaimlock(string keyinfo)`

This method operates the same as the **`readaim`** method, except that the record is locked.

`public int readaim(string[] keyinfoarray)`

This method reads the first record of a result set of records into the record buffer. The result set is created using the AIM index access method. The **`keyinfoarray`** parameter array specifies the multiple lookup criteria. The syntax of each of the lookup criteria is as specified above. The result set consists of those records that match all of the lookup criteria.

`public int readaimlock(string[] keyinfoarray)`

This method operates the same as the **`readaim`** method, except that the record is locked.

`public void writerandom(int reclen, int recnum)`

This method writes a record from the record buffer using the random access method. The length of the record written is specified by the **`reclen`** parameter. The record is written into the record number specified by the **`recnum`** parameter.

`public void writenext(int reclen)`

This method writes a record from the record buffer using the sequential access method. The length of the record written is specified by the **`reclen`** parameter. The record is written after the most recently written record.

`public void writeateof(int reclen)`

This method writes a record from the record buffer using the sequential access method. The length of the record written is specified by the **`reclen`** parameter. The record is written at the end of the file.

`public void writekey(int reclen, string key)`

This method writes a record from the record buffer using the sequential access method and the indexed access method. The length of the record written is specified by the **`reclen`** parameter. The record is written at the end of the file. The value of the **`key`** parameter and the position of the record just written is added to the index file.

`public void writeaim(int reclen)`

This method writes a record from the record buffer using the sequential access method and the AIM indexed access method. The length of the record written is specified by the **`reclen`** parameter. The record is written at the end of the file. The AIM index is updated to reflect the values of the AIM keys that were contained in the record just written.

`public void insertkey(string key)`

This method inserts a key specified by **`key`** and the current file position into the index file.

`public void insertkeys(int reclen)`

This method updates the AIM index to reflect the values of the AIM keys that are contained in the record in the record buffer.

`public void update(int reclen)`

This method updates a record from the record buffer. The length of the record written is specified by the **`reclen`** parameter. The record is written over the record at the current file position.

`public void delete()`

This method deletes the record that was most recently read or written.

`public void delete(string key)`
This method deletes a key from the index file and the record that it is associated with from the data file. The **`key`** parameter specifies the key that is deleted.

`public void deletekey(string key)`
This method deletes the key that is specified by the **`key`** parameter from the index file.

`public unlock(long pos)`
This method unlocks the record at the file position specified by the **`pos`** parameter.

`public void unlock()`
This method unlocks records locked by this instance of **`File`**.

`public void weof(long pos)`
This method writes an end of file marker at the position specified by the **`pos`** parameter.

`public int compare(dbcfs.File file)`
This method compares this **`File`** instance with the instance specified by the file parameter. The integer value that is returned is zero if the files are the same. The value will be less than zero if the files compare such that this **`File`** instance is less than the **`File`** instance of the file parameter. The value will be positive otherwise. The concept of less than is not defined, but will always be the same for comparison of the same files. This method is used for ordering **`filelock`** calls to avoid deadlock.

`public void filelock()`
This method locks this data file. If the data file is already locked, this method will wait for the data file to be unlocked by the other program.

`public void fileunlock()`
This method unlocks this data file.

5.4 Util

The **`Util`** class implements several miscellaneous static methods that are used by the **`Connection`** and **`File`** classes and may also be used by other C# programs.

`public static string currentdatetime()`
This method returns a value that is 16 characters long which contains the current timestamp in YYYYMMDDHHMMSSHH format (the final HH is hundredths of a second).

`public static string itos(int value, int size)`

This method returns a value that is the string value of the **`value`** parameter and is the length specified by the size parameter. The value is truncated or blank filled on the left to fit the length specified.

`public static string itoszf(int number, int count)`

This method returns a value that is the integer value of the value parameter and is the length specified by the **`count`** parameter. The value is truncated or zero filled on the left to fit the length specified.

`public static string stos(string value, int size)`

This method returns a value that is the truncated or blank filled on right value of the **`value`** parameter. The length of the string value returned is specified by the **`size`** parameter.

`public static string stonums(string value, int size1, int size2)`

This method returns a value that is the reformatted numeric value specified by the **`value`** parameter. The **`size1`** parameter specifies the number of digits left of the decimal point in the result. The **`size2`** parameter specifies the number of digits right of the decimal point in the result. If **`size2`** is zero, no decimal point is contained in the resulting value. The value is truncated, not rounded, if necessary.

`public static string zeronums(int size1, int size2)`

This method returns a value that is numeric value zero with a format specified by the **`size1`** and **`size2`** parameters. If **`size2`** is zero, **`size1`** specifies the number of characters in the resulting string, which will contain the digit zero right justified and preceeded by blanks. If **`size2`** is non-zero, **`size1`** specifies the number of leading blanks, followed by a decimal point and then by **`size2`** zeroes.

`public static void itoca(int value, char[] dest, int destoff, int size)`

This method converts an integer value specified by the value parameter to a string of characters that are stored in successive characters of the **`dest`** parameter array. The characters are stored starting at the index specified by the **`destoff`** parameter. The number of characters stored is specified by the **`size`** parameter. The characters are truncated or blank filled on the left in the same manner as by the **`itos`** method.

`public static int catoi(char[] source, int sourceoff, int size)`

This method returns an integer that is the value specified by the characters from the source array, starting with the character at the index specified by the sourceoff parameter. The number of characters used is specified by the **`size`** parameter.

5.5 ConnectionException

The **`ConnectionException`** class implements a class that is used for exceptional conditions by the **`Connection`** class. The **`ConnectionException`** class extends **`Exception`**.

`public string getinfo()`

This method returns a value that contains descriptive information about this **`ConnectionException`**.

`public string ToString()`

This method returns a string that contains descriptive information about this **`ConnectionException`**.

5.6 FileException

The **`FileException`** class implements a class that is used for exceptional conditions by the **`File`** class. The **`FileException`** class extends **`Exception`**.

`public FileException(int error)`

This FileException constructor is used when only an error number is available.

`public FileException(int error, string info)`

This FileException constructor is used when an error number and descriptive information are available.

`public int geterror()`

This method returns the error number associated with this FileException.

`public string getinfo()`

This method returns a value that contains descriptive information about this **`FileException`**.

`public string ToString()`

This method returns a string that contains the error number and descriptive information about this **`FileException`**.

Chapter 6. C API

The C client programming application programming interface provides several C functions that allow access to DB/C FS data files using traditional sequential, random, indexed and AIM indexed access methods. These functions are provided in the file **`fsfileio.c`**. The **`fsfileio.h`** include file contains the function prototypes and defininitions for what follows. The C functions require TCP/IP code which is contained in **`tcp.c`** and **`tcp.h`**. You may need to modify the TCP/IP interface for other platforms.

The type **`OFFSET`** is a 64 bit signed integer.

By default, these functions require the OpenSSL encryption library to be included. OpenSSL can be obtained at www.openssl.org. If you do not require encryption, compile **`tcp.c`** with the **`-DDBC_SSL=0`** command line option.

Unless otherwise specified, the value zero is returned if the function completes successfully.

```
int fsgetgreeting(char *server, int port, int encrypt, char *auth, char *msg, int length)
```

If the DB/C FS server is available, the **`fsgetgreeting`** function returns a zero-delimited string that is the version number of the DB/C FS server. The **`server`** parameter is a zero-delimited string that is either the domain name or the TCP/IP address of the computer running the DB/C FS server. The **`port`** parameter is the TCP/IP port number of the DB/C FS server. If port is zero, the default value is used. If **`encrypt`** is non-zero, encryption is used. The **`auth`** parameter should be set to null. The **`msg`** parameter points to the area to which the return string is moved. The **`length`** parameter specifies the size of the **`msg`** memory. The value -1 is returned if an error occurs.

```
int fsgeterror(char *msg, int length)
```

The **`fsgeterror`** function returns a zero delimited string that describes the most recent error encountered by any of the functions described in this chapter.

```
int fsconnect(char *server, int serverport, int localport, int encrypt, char *authfile,
     char *database, char *user, char *pswd);
```

The **`fsconnect`** function creates a connection between this program and the DB/C FS server. The **`server`** parameter is a zero-delimited string that is either the domain name or the TCP/IP address of the computer running the DB/C FS server. The **`serverport`** parameter is the TCP/IP port number of the DB/C FS server. If **`serverport`** is zero, the default value is used. The **`localport`** parameter is the local TCP/IP port number that the server will connect back to to establish the connection. If this value is –1, a dynamically created port number will be used. If this value is zero, then this signals the server that the connection should use the **sport** feature of the server. If encrypt is non-zero, then messages sent and received will be encrypted. The authfile parameter should be set to null. The **database** parameter is a zero-delimited string that is the name of the database to be accessed which is the name of the DBD file without the **.dbd** extension). The **user** and **pswd** parameters point to zero-delimited strings that are the user id and password reuired for login to the DB/C FS server. If the value returned is positive, the connec tion was created successfully and the connection handle is the value returned. If the value returned is negative, the connection was not created.

```
int fsdisconnect(int connection);
```

The **fsdisconnect** function destroys the connection with the DB/C FS server. The **connection** parameter is the connection handle that was returned by the fsconnect function. If the **disconnect** function fails, -1 is returned.

```
int fsopen(int connection, char *txtfilename, int options, int recsize);

int fsprep(int connection, char *txtfilename, int options, int recsize);

int fsopenisi(int connection, char *isifilename, int options, int recsize, int keysize);

int fsprepisi(int connection, char *txtfilename, char *isifilename, int options,
     int recsize, int keysize);

int fsopenaim(int connection, char *aimfilename, int options, int recsize);

int fsprepaim(connection, char *txtfilename, char *aimfilename, int options,
     int recsize, char *keyinfo, char matchchar);
```

The **open** and **prep** functions open and create files. The **fsopen** function opens a text file for sequential and random access. The **fsprep** function creates a text file for sequential and random access. The **fsopenisi** function opens an indexed file and a text file for indexed sequential access. The **fsprepisi** function creates an indexed file and a text file for indexed sequential access. The **fsopenaim** function opens an associative indexed file and a text file for associative index access. The **fsprepaim** function creates an associative indexed file and a text file for associative index access.

The **connection** parameter is the connection handle that was returned by the **fsconnect** function.

The **txtfilename** parameter is the pointer to a zero-delimited string that specifies the text file that is to be opened or created.

The **isifilename** parameter is the pointer to a zero-delimited string that specifies the index file that is to be opened or created.

The **aimfilename** parameter is the pointer to a zero-delimited string that specifies the associative index file that is to be opened or created.

The **`options`** parameter specifies several file open and create options. Here are the **`#defines`** for the values for options that may be or'd together.

```
FS_VARIABLE
FS_COMPRESSED
FS_TEXT
FS_DATA
FS_CRLF
FS_BINARY
FS_READONLY
FS_EXCLUSIVE
FS_DUP
FS_NODUP
FS_CASESENSITIVE
FS_LOCKAUTO
FS_LOCKSINGLE
FS_LOCKNOWAIT
FS_CREATEONLY
```

The **`recsize`** parameter specifies the maximum record size for variable length records and the record size for fixed length records. The **`keysize`** parameter specifies the index key size. The **`keyinfo`** parameter is the pointer to a zero-delimited string that specifies the aim key information. Its format is the same as in DB/C DX. The **`matchchar`** parameter specifies the aim search character.

If the value returned is a positive number, then the open or create was successful and the value is the file handle. If the value returned is negative, then an error occurred.

```
int fsclose(int filehandle);
```

The **`fsclose`** function closes the file handle specified. The value -1 is returned if an error occurs.

```
int fsclosedelete(int filehandle);
```

The **`fsclosedelete`** function closes the file handle specified and deletes the associated file or files. The value -1 is returned if an error occurs.

```
int fsfposit(int filehandle, OFFSET pos);
```

The **`fsfposit`** function returns the value called the current file position into the memory pointed to by the **`pos`** parameter. The current file position is the offset (in bytes) of the last record read from or written to the text file, or is the value specified by the

`reposit` function if it was called after the most recent read or write. The initial value of **`fposit`** immediately after a file is opened is zero. The return value -1 is if the file handle is invalid.

```
int fsreposit(int filehandle, OFFSET pos);
```

The **`fsreposit`** function sets the value of the current file position to the value specified by the **`pos`** parameter. The value returned indicates what the position is. The return value 0 means the current file position was set before end of file. The return value 1 means the current file position was set at end of file. The return value 2 means the current file position was not set because the value specified was past end of file. The return value -1 is if the file handle is invalid.

```
int fssetposateof(int filehandle);
```

The **`fssetposateof`** function sets the current file position to the end of the text file. The return value -1 is if the file handle is invalid.

```
int fsreadrandom(int filehandle, int recnum, char *record, int length);

int fsreadrandomlock(int filehandle, int recnum, char *record, int length);

int fsreadnext(int filehandle, char *record, int length);

int fsreadnextlock(int filehandle, char *record, int length);

int fsreadprev(int filehandle, char *record, int length);

int fsreadprevlock(int filehandle, char *record, int length);

int fsreadsame(int filehandle, char *record, int length);

int fsreadsamelock(int filehandle, char *record, int length);

int fsreadkey(int filehandle, char *key, char *record, int length);

int fsreadkeylock(int filehandle, char *key, char *record, int length);

int fsreadkeynext(int filehandle, char *record, int length);

int fsreadkeynextlock(int filehandle, char *record, int length);

int fsreadkeyprev(int filehandle, char *record, int length);

int fsreadkeyprevlock(int filehandle, char *record, int length);

int fsreadaimkey(int filehandle, char *aimkey, char *record, int length);
int fsreadaimkeylock(int filehandle, char *aimkey, char *record, int length);

int fsreadaimkeys(int filehandle, char **aimkeys, int aimkeyscount, char *record,
     int length);

int fsreadaimkeyslock(int filehandle, char **aimkeys, int aimkeyscount, char *record,
     int length);

int fsreadaimnext(int filehandle, char *record, int length);

int fsreadaimnextlock(int filehandle, char *record, int length);

int fsreadaimprev(int filehandle, char *record, int length);

int fsreadaimprevlock(int filehandle, char *record, int length);
```

The various **fsread** functions read a record into the record memory area. The **filehandle** parameter is the file handle that was returned by an **fsopen** or **fsprep** function. The **record** parameter points to an area that will contain the record after it is read. The **length** parameter is the size of the record memory. If the value returned by an **fsread** function is non-negative, then it is the number of bytes contained in the record that was read and stored in the record memory area. Note that zero is a valid record length. If the value returned by an **fsread** function is -1, then the **filehandle** is invalid. If the value returned by fsread is -3, then the record was not found or the file position was at the end of the file. If the value returned by an **fsread** function is -2, then the record lock could not be obtained and **LOCKNOWAIT** was specified. Any other negative return value indicates an error.

For each of the following descriptions, the **fsread**...**lock** function operates the same as the corresponding function without the lock suffix, except that the record is locked.

The **fsreadrandom** function reads a fixed length record at the record number specified by **recnum**. The **recnum** value zero indicates the first record in the file.

The **fsreadnext** function reads a record using the sequential access method to retrieve the record that is after the record most recently read.

The **fsreadprev** function reads a record using the sequential access method to retrieve the record that is before the record most recently read.

The **fsreadsame** function reads the record at the current file position.

The **fsreadkey** function reads the record using the indexed sequential access method. The **key** parameter is a pointer to a zero-delimited string that specifies the key used to lookup the record.

The **`fsreadkeynext`** function reads the record using the index sequential access method to read the next key sequential record.

The **`fsreadkeyprev`** function reads the record using the index sequential access method to read the previous key sequential record.

The **`fsreadaimkey`** function reads the record using the associative index access method to read the first record that matches the criteria specified by the **`aimkey`** parameter. The **`aimkey`** parameter is a pointer to a zero-delimited string of 4 or more characters. The first two characters are digits (blank filled in the first position) that specify the column for the search. The third character is **`F`** or **`X`** or **`L`** or **`R`** for floating, exact, left or right match, respectively. The remaining characters are the match characters.

The **`fsreadaimkeys`** function reads the record using the associative index access method to read the first record that matches the criteria specified by the **`aimkeys`** and **`aimkeyscount`** parameters. The **`aimkeys`** parameter is a pointer to an array of pointers, each of which points to a key in the same format as described in the previous paragraph. The **`aimkeyscount`** parameter specifies the number of entries in the array of pointers.

The **`fsreadaimnext`** function reads the next record that matches the key(s) specified by the most recent **`fsreadaimkey`** or **`fsreadaimkeys`** function.

The **`fsreadaimprev`** function reads the previous record that matches the key(s) specified by the most recent **`fsreadaimkey`** or **`fsreadaimkeys`** function.

```
int fswriterandom(int filehandle, int recnum, char *record, int length);

int fswritenext(int filehandle, char *record, int length);

int fswriteateof(int filehandle, char *record, int length);

int fswritekey(int filehandle, char *key,char * record, int length);

int fswriteaim(int filehandle, char *record, int length);
```

The **`fswriterandom`** function writes a record using the random access method. The **`record`** parameter points to the record. The length of the record is specified by the **`length`** parameter. The record is written into the record number specified by the **`recnum`** parameter.

The **`fswritenext`** function writes a record using the sequential access method. The **`record`** parameter points to the record. The length of the record is specified by the **`length`** parameter. The record is written after the most recently written record.

The **`fswriteateof`** function writes a record using the sequential access method. The **`record`** parameter points to the record. The length of the record is specified by the **`length`** parameter. The record is written at the end of the file.

The **`fswritekey`** function writes a record using the sequential access method and the indexed access method. The **`record`** parameter points to the record. The length of the record is specified by the **`length`** parameter. The record is written at the end of the file. The value of the **`key`** parameter and the position of the record just written is added to the index file.

The **`fswriteaim`** function writes a record using the sequential access method and the AIM indexed access method. The **`record`** parameter points to the record. The length of the record written is specified by the **`length`** parameter. The record is written at the end of the file. The AIM index is updated to reflect the values of the AIM keys that were contained in the record just written.

```
int fsinsertkey(int filehandle, char *key);
```

The **`fsinsertkey`** function inserts a key and the current file position into an index file. The key parameter is a pointer to a zero-delimited string that is the key inserted.

```
int fsinsertkeys(int filehandle, char *record, int length);
```

The **`fsinsertkeys`** function causes the aim index to be updated to reflect the values of the AIM keys that are contained in the record specified by the **`record`** and **`length`** parameters.

```
int fsupdate(int filehandle, char *record, int length);
```

The **`fsupdate`** function updates a record in place. The record written is specified by the **`record`** and length parameters. The new record is written over the record at the current file position.

```
int fsdelete(int filehandle);
```

The **`fsdelete`** function deletes the record that was most recently read or written.

```
int fsdeletekey(int filehandle, char *key);
```

The **`fsdeletekey`** function deletes a key from the index file and the record that it is associated with from the data file. The **`key`** parameter is a pointer to a zero-delimited string that specifies the key that is deleted. If the key is not found in the index file, then a value of one is returned.

```
int fsdeletekeyonly(int filehandle, char *key);
```

The **`fsdeletekeyonly`** function deletes only a key from the index file. The **`key`** parameter is a pointer to a zero-delimited string that specifies the key that is deleted. If the key is not found in the index file, then a value of one is returned.

```
int fsunlock(int filehandle, OFFSET pos);
```

The **fsunlock** function unlocks the record at the file position specified by the **pos** parameter.

```
int fsunlockall(int filehandle);
```

The **fsunlockall** function unlocks all locked records.

```
int fsweof(int filehandle, OFFSET pos);
```

The **fsweof** function writes an end of file marker at the position specified by the **pos** parameter.

```
int fscompare(int filehandle, int filehandle2);
```

The **fscompare** function provides a repeatable comparison between the files associated with the two file handles. The integer value that is returned is zero if the files are the same. The value will be one if the files compare such that the first filehandle is considered less than the second file handle. The value will be two otherwise. The concept of less than is not defined, but will always be the same for comparison of the same files. This function is used for ordering **fsfilelock** calls to avoid a deadlock.

```
int fsfilelock(int filehandle);
```

The **fsfilelock** function locks the file. If the data file is already locked, this function will wait for the data file to be unlocked by the other program.

```
int fsfileunlock(int filehandle);
```

The **fsfileunlock** function unlocks this data file.

Chapter 7. Programming Interface Error Codes

Error Code	Meaning
601	File not found during OPEN, RENAME, or ERASE
602	Open mode conflicts with another program
603	Null or invalid file name used with OPEN, PREPARE, RENAME, or ERASE
604	Invalid file type used with OPEN, PREPARE, RENAME, or ERASE
605	Attempt to PREPARE or RENAME a file that already exists
606	Key length specified with OPEN differs from index file
607	Access violation during OPEN, PREPARE, RENAME, or ERASE
608	Associative key specification in PREPARE missing or invalid
609	Index key specification invalid in PREPARE
610	Invalid record length specified in OPEN or PREPARE
611	Attempt to rename a file to a different resource
612	Error during close
613	Unable to create file during PREPARE
614	Out of memory during OPEN, PREPARE, RENAME, or ERASE
701	File not open; operation attempted on closed file
702	Attempt to WRITE to read-only file
703	Attempt to randomly access a variable length record file; only a fixed length record file can be randomly accessed
704	Invalid tab value used with READ, UPDATAB, or WRITAB
705	Attempt to access a file that is off-line or otherwise not available

706	Error during WRITE
707	Access violation during READ, WRITE, or WRITAB
708	Attempt to WRITE or INSERT with null index key
709	Attempt to WRITE or INSERT a duplicate key
710	Attempt to update to invalid file position
711	Attempt to update a compressed file
712	Associative read key specification error
713	Associative read insufficient key(s)specified
714	Invalid associative index position for READKG or READKGP
715	Attempt to read a record that is too small
716	Attempt to read, write or update a record that is too large
717	Invalid data file position for INSERT or DELETE
718	Attempt to perform an indexed READ with null key and invalid position
719	Attempt to INSERT two identical keys for the same record
720	Attempt to READ a record that does not exist or was deleted
721	Specified key too long in READ, WRITE, INSERT, DELETE or DELETEK

Appendix A. Sample DBD File

Here is a sample DBD file:

```
<dbcfsdbd>
     <access>ALL</access>
     <table>
          <name>IM</name>
          <description>Item Master File</description>
          <textfile>ITEM.TXT</textfile>
          <textfiletype>standard</textfiletype>
          <indexfile>ITEM1.ISI</indexfile>
          <indexfile>ITEM2.ISI</indexfile>
          <indexfile>ITEM3.ISI</indexfile>
          <indexfile>ITEM.AIM</indexfile>
          <fixedlength/>
          <columns>
               <c>
                    <n>ITEM</n>
                    <t>CHAR(6)</t>
                    <d>Our item number</d>
               </c>
               <c>
                    <n>VEND</n>
                    <t>CHAR(6)</t>
                    <d>Vendor code</d>
               </c>
               <c>
                    <n>VITM</n>
                    <t>CHAR(12)</t>
                    <d>Vendor item number</d>
               </c>
               <c>
                    <n>CLAS</n>
                    <t>CHAR(4)</t>
                    <d>Item class</d>
               </c>
```

```
<c>
     <n>SEQN</n>
     <t>CHAR(3)</t>
     <d>Sequence number</d>
</c>
<c>
     <n>MNFR</n>
     <t>CHAR(30)</t>
     <d>Manufacturer</d>
</c>
<c>
     <n>MITM</n>
     <t>CHAR(15)</t>
     <d>Manufacturer item number</d>
</c>
<c>
     <n>DESC</n>
     <t>CHAR(60)</t>
     <d>Description</d>
</c>
<c>
     <n>LIST</n>
     <t>NUM(9,3)</t>
     <d>List price</d>
</c>
<c>
     <n>COST</n>
     <t>NUM(9,3)</t>
     <d>Cost</d>
</c>
<c>
     <n>LPDT</n>
     <t>CHAR(6)</t>
     <d>Last purchase date</d>
</c>
<c>
     <n>LPPR</n>
```

```
                    <t>NUM(9,3)</t>
                    <d>Last purchase price</d>
               </c>
               <c>
                    <n>HAND</n>
                    <t>NUM(6,0)</t>
                    <d>On hand</d>
               </c>
               <c>
                    <n>QTYO</n>
                    <t>NUM(6,0)</t>
                    <d>On order</d>
               </c>
          </columns>
     </table>
     <table>
          <name>VM</name>
          <description>Vendor Master File</description>
          <textfile>VEND.TXT</textfile>
          <indexfile>VEND.ISI</indexfile>
          <fixedlength/>
          <columns>
               <c>
                    <n>VEND</n>
                    <t>CHAR(6)</t>
                    <d>Vendor code</d>
               </c>
               <c>
                    <n>NAME</n>
                    <t>CHAR(30)</t>
                    <d>Vendor name</d>
               </c>
          </columns>
     </table>
</dbcfsdbd>
```

Appendix B. Log File Format

The change logging feature of DB/C FS is implemented at the data record level. This feature logs all write, update and delete activity for files that are changed via both SQL and file I/O interfaces. Information about indexes (.isi and .aim) is not logged; it can be inferred from the value of the records being updated and written. Logging information from all connections is written into a single log file. Each change (write, update or delete) is written to the log file as a single XML element.

Log information is written to the log file whose name is specified in the configuration file. When the active log file is changed to an archive log file, it is renamed and a new log file is created while DB/C FS is running. This change happens either by running **dbcfs** with the **-g** option, or by running **dbcfsadm** with the **-g** option. **dbcfsadm** can be executed using the UNIX cron command or similar to automatically create archives.

The archive log file name is the same as the log file name except it has an underscore (_) followed by a 16 digit timestamp appended to the end of the file name. The file extension will be the same.

The following XML elements are written to the log file:

`<start>`*timestamp***`</start>`** (at **dbcfs** startup, **dbcfsadm –g** or **dbcfsadm –l**)

`<stop>`*timestamp***`</stop>`** (at **dbcfs** shutdown, **dbcfsadm –g** or **dbcfsadm -x** if logging is active)

```
<connect>
    <c>connection handle</c>
    <user>user name</user>
    <database>database name</database>
</connect>
```

```
<disconnect>
    <c>connection handle</c>
</disconnect>
```

```
<open>
    <c>connection handle</c>
    <f>file handle</f>
    <name>data file name</name>
    <user>user name</user>
    <stamp>time stamp</stamp>

</open>

<close>
    <f>file handle</f>
    <name>data file name</name>
    <user>user name</user>
    <stamp>time stamp</stamp>
</close>

<w>
    <f>file handle</f>
    <name>data file name</name>
    <user>user name</user>
    <stamp>time stamp</stamp>
    <n>new data written to end of file</n>
</w>

<d>
    <f>file handle</f>
    <name>data file name</name>
    <user>user name</user>
    <stamp>time stamp</stamp>
    <o>data that is deleted</o>

</d>

<u>
    <f>file handle</f>
    <name>data file name</name>
    <o>old data that is over written</o>
    <user>user name</user>
    <stamp>time stamp</stamp>
    <n>new data that is over writing the old data</n>
</u>
```

The log file will contain a continuous stream of data with no record delimiters.