

[print](#) | [close](#)

APIs by Example: Analyzing Logical Files Using the QDBRTVFD File API

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 05/27/2010 (All day)

The Retrieve Database File Description (QDBRTVFD) API is by far one of the most challenging and discouraging APIs that ever left IBM's drawing board alive. But because it is also an extremely powerful API, I keep coming back to it whenever database files are involved in the equation. There's no detail in the complex and nested structure that makes up a file object that is out of reach for the QDBRTVFD API. Whether physical files, logical files in various flavors, any kind of file field information, key information—you name it, it's all there—buried somewhere in the many file description layers and structures exposed by this API.

Today's API by Example uses the QDBRTVFD API to catch and eliminate the dangers and problems potentially caused by misplaced logical files. While there can be valid reasons for a logical file to be located in a library different from a physical file it is based on, it can also sometimes be traced back to a mistake or misunderstanding on the part of the developer creating the logical file. Should that situation arise, the Analyze Logical File Integrity (ANZLFITG) command will help you quickly identify and correct the problem.

If you don't have a completely waterproof separation of your production and development environment—whether physically or by means of resource security—as well as a thoroughly verified procedure for moving objects between one and the other, there's a risk that you at times for whatever reason might end up with logical files in test libraries pointing to a physical file in a production library or vice versa. Subsequently running production programs might perform updates in partial against test data, or likewise, testing programs under development in turn will update your production data. Encountering such situations will in a worst-case scenario keep you occupied for quite a while before you have reestablished the database integrity.

Another issue originating from logical files located in libraries different from their physical files' libraries relates to save and restore dependencies. At releases earlier than 6.1, it's impossible to restore a logical file if a based-on physical file doesn't exist in the library in which it was located when the logical file was saved. So if the logical file library is restored before the physical file library, the restore will fail. Given, for example, a disaster recovery situation, failing to comply with this restriction can add significantly to an already tense atmosphere.

I've added a link below to the information about the new *restore deferred object* option enabling you to restore logical files associated with currently nonexistent physical files at release 6.1 and later.

Whatever the potential problem is, the ANZLFITG command will help you detect it promptly or, in case logical and physical file library differences are by design, document the issue well ahead of situations in which such information is vital in order to maintain the control and initiative, and provide the foundation for timely planning of such endeavors. Before we look at how to perform a

logical file analysis tracking potential culprits, however, let's have a look at the rules determining where the based-on physical files are found when creating logical files.

If you run the Create Logical File (CRTLF) file command and the physical files are specified unqualified on the PFILE DDS-keyword, the library list is used to identify the physical file. So before running or submitting the CRTLF command, due diligence is essential, as far as controlling and verifying the library list is concerned. So if you create a logical file in library XYZ, even though the based-on physical file exists in library XYZ, there's no guarantee that this file will be used. If the physical file also exists in a library ahead of XYZ in the library list of the job compiling the logical file, for example library ABC, the physical file in library ABC will be used.

As for the Create Duplicate Object (CRTDUPOBJ) command, the rules are a bit more complex. When you run the CRTDUPOBJ command for a logical file two rules apply, depending on whether the logical file being copied is associated with a physical file residing in the same library as the original logical file.

1. If the logical file being copied resides in the same library as the physical file it is based on, a duplicate of the based-on physical file must at the time of the copy exist in the target library. Following the completion of the copy command, the new logical file will be based on the physical file located in the target library. The CRTDUPOBJ command this way enforces that the new logical file remains associated with a physical file residing in a common library.
2. If, however, the logical file being copied is based on a physical file in a different library than itself, the new logical file will be based on the same physical file as the logical file being copied. Not being specifically acquainted with this rule might cause someone to anticipate another behavior, so this situation is the one most likely to generate a potential future problem, if caution is not exercised.

For unqualified table names in SQL CREATE VIEW statements, the following rules apply to determine which table is actually being referenced:

1. If the unqualified name corresponds to one or more common table expression table-identifiers specified in the fullselect, the name identifies the common table expression that is in the innermost scope.
2. Otherwise, the name identifies a persistent table, a temporary table, or a view that is present in the default schema. The default schema for system naming is *LIBL, for SQL naming USER, and can be set using the SET SCHEMA statement.

These are the main rules to observe. Please refer to the respective manuals for more details. I've included relevant links at the end of this article. Before I go through the process of retrieving file information, let's have a look at the ANZLFITG command prompt to decide exactly what information is needed for the command processing program (CPP) to do its job:

```
Analyze Logical File Integrity (ANZLFITG)

Type choices, press Enter.
```

Logical file	*ALL	Name, generic*,
*ALL		
Library	*LIBL	Name, *LIBL,
*CURLIB...		
Include:		
Join logical files	*YES	*YES, *NO
Multi format logical files . .	*YES	*YES, *NO
IBM libraries	*YES	*YES, *NO
Violation types	*ALL	*ALL, *PFLIB,
*LFLIB		
Sort order	*FILE	*FILE, *LIB
Output	*	*, *PRINT

The list of logical file candidates to be examined is defined by the command's main parameter, allowing you to specify a generic name or the special value `*ALL` as the file name, and a number of library special values as well as a library name to qualify the specified file name as required. The CPP uses the Open List of Objects (QGYOLOBJ) API to retrieve this list and then for each of the logical files returned in the object list, the QDBRTVFD API to extract the file information necessary to perform the evaluation of the INZLFITG command's selection criteria as well as the file details to be shown on the command's list panel.

The *Include* parameter allows you to specify three list filter criteria, which effectively let you decide whether to include the following types of logical files: Join logical files, multiformal logical as well as logical files residing in an IBM library, the latter being defined by an object creator user profile of `*IBM`. This can be quite easily accessed using the Retrieve Object Description (QUSROBJD) API against the logical file library object. The two other criteria I'll get back to in a moment.

The *Violation type* parameter specifies the type of logical file integrity violation to include in the list. Type `*LFLIB` means that a logical file is based on a physical file located in another library than the logical file; however the physical file does not exist in the logical file library. This type of violation is less error indicative than type `*PFLIB`, which covers the situation in which a logical file points to a physical file in another library than the logical file, while a physical file *does* exist in the logical file library. This is of course a strong indication that something probably is not in accordance with the intention of the programmer who created the logical file. You have the option of including only one of the violation types, or both.

To sum it all up, I will need to extract and evaluate the following information in order to produce the desired logical file list:

- The logical file library
- The physical file library for each based-on physical file
- The logical file type (join logical, multiformal logical)
- The logical file creator user profile

- Whether physical file exists in logical file library

The logical file library is straightforward; I'll get that as part of the object information data structure returned by the QGYOLOBJ API for each object listed. For the based-on physical file library, I'll have to resort to the QDBRTVFD API. Another alternative to obtain this information would be the Retrieve Member Description (QUSRMBRD) API, but I've on previous occasions found it to return incomplete information as far as the file record format name is concerned which I'll need for the list panel, so I'll stick to good old QDBRTVFD.

As discussed in earlier articles involving the QDBRTVFD API and to which I've included links at the end of this article, the API returns one of four groups of definition or information templates related to a file object:

- FILDo100 File definition template
- FILDo200 Format definition template
- FILDo300 Key field information template
- FILDo400 Trigger information template

For the purpose at hand, the FILDo100 File definition template is the appropriate choice. In the QDBRTVFD API documentation, there's diagram describing the FILDo100 definition template's many structures and their respective correlations. As you will note, the place to start is at the top with the *File Definition Header* structure Qdb_qdbfh (FDT). This structure contains file-level attributes as well as the offsets to the next layer of detail information structures. To find out which offsets I'm going to use, I'll need to identify which substructures hold the information that I'm after.

Because the aforementioned structure diagram doesn't immediately identify where the based-on physical file information is located, it takes a little research to discover that these details are found in the *File Scope Array* (Qdb_Qdbfb) structure. In the header of the Qdb_Qdbfb structure description section, there's also a reference to where you can locate the offset to this structure. As it turns out, it's the offset field Qdbfos in the FDT header section, Qdb_qdbfh. The offset is calculated from the beginning of the FDT header section.

Since I use pointer arithmetic to resolve the substructures, it's very fast and simple to populate the Qdb_Qdbfb structure. I base the Qdb_Qdbfb structure on a space pointer, and set the value of this pointer to the address of the beginning of the FDT header section, and then add the offset found in the Qdbfos field, as in the following example in which the pQdb_qdbfh pointer contains the address of the beginning of the Qdb_qdbfh structure:

```
pQdb_Qdbfb = pQdb_qdbfh + Qdb_qdbfh.Qdbfos;
```

Following the execution of the above statement, pointer pQdb_Qdbfb now points to the first byte of information in the File Scope Array returned by the QDBRTVFD API. The array part of the structure name indicates that there's a multiple of structures, for logical files one for each based-on physical file. You get to the next array entry by adding the size of the Qdb_Qdbfb structure to the pQdb_Qdbfb pointer and in order to process all array entries you repeat to do so for as many times as defined by the Qdbflbnum field also found in the FDT header section and defining the number of based-on physical file record formats for a logical file:

```
For   Idx = 1   To   Qdb_qdbfh.Qdbflbnum;
```

```
// Perform file scope array entry processing here
```

```
If Idx
```

The information I need is found in the `Qdbfbf` field containing the physical file name, the `Qdbfbfl` file holding the physical file library name, and the `Qdbft` field defining the record format name. Here's an excerpt of the `Qdb_Qdbfb` structure definition in the CPP:

```
**-- File scope array:
```

```
D Qdb_Qdbfb      Ds      Qualified Based( pQdb_Qdbfb )

D  Reserved_48      48a

D  Qdbfbf_q         20a

D  Qdbfbf           10a  Overlay( Qdbfbf_q: 1 )

D  Qdbfbfl          10a  Overlay( Qdbfbf_q: 11 )

D  Qdbft            10a

D  Reserved_49      37a

D  Qdbfbgky         5i 0

D  Reserved_50      2a

D  Qdbfblk          5i 0
```

```
...
```

Given the mentioned `Qdbflbnum` field defining the number of based-on physical file record formats, which allows me to identify multiformat logical files, at this point it only remains for me to establish whether a logical file is also a join logical file, in order to provide for the ANZLFITG command's two related include parameters. A quick search of the QDBRTVFD API documentation locates the Logical File Specific Attributes (`Qdb_Qdbflogl`) structure, which holds a *Join logical file indicator*.

To locate the `Qdb_Qdbflogl` structure, the documentation points to the `Qdbflfof` field in the FDT header section, `Qdb_Qdbfh`. The offset is as usual calculated from the beginning of the FDT header section. The only problem is that there's no `Qdbflfof` field to be found in the `Qdb_Qdbfh` structure, or elsewhere for that matter. Scanning for the `Qdb_Qdbflogl` structure name, however, quickly reveals that the offset field is named `Qdblfof` instead. So prior to accessing the data in the `Qdb_Qdbflogl` structure, I set its based-on pointer to the address found by pointer arithmetic along the lines specified above:

```
pQdb_Qdbflogl = pQdb_Qdbfh + Qdb_Qdbfh.Qdblfof;
```

The Join logical file indicator is present in the form of a bit field, which makes it a little more convoluted to access. According to the documentation, the indicator is located as the third leftmost bit in the Qdb_Qdbflogl structure's Qlfa field. I use the tstbts (Test bit in string) C library function to set the Qdbfjoin integer to either zero or one:

```
Qdbfjoin = tstbts( %Addr( Qdb_Qdbflogl.Qlfa ): TYP_JOIN );
```

The tstbts() function counts the leftmost bit as zero, the next as one, and so on, so to check for the bit value of the third leftmost bit you need to specify 2 as the second argument for the tstbts() function. I use the named constant TYP_JOIN defined as the digit 2 to help me document this significance of the digit. If you'd rather stick to RPG/IV built-in functions, you can achieve the same result using the %BitAnd() Bitwise AND Operation, specifying a bit-mask with the desired bit set on as the function argument:

```
D TYP_JOIN          C          x'20'

      If %BitAnd( Qdb_Qdbflogl.Qlfa: TYP_JOIN )  x'00';
          Qdbfjoin = 1;
      Else;
          Qdbfjoin = 0;
      EndIf;
```

Anyway, at this point I have all the main building blocks that I need in order to put together the ANZLFITG CPP. Despite my somewhat intimidating introduction to the QDBRTVFD API in the opening section of this article, getting at the information needed for this utility was easy. Once you get used to the ideas and conventions practiced by the QDBRTVFD API, any type of database file information is within your reach. You can follow the program flow and sequence of events by running the CPP in your favorite source debugger and see how the pieces (hopefully) fit together. I tried running the following command on my system:

```
ANZLFITG FILE(*ALL/*ALL)
        INCLUDE(*YES *YES *YES)
        VIOLTYPE(*ALL)
        ORDER(*LIB)
        OUTPUT(*)
```

Listing and processing all files on a system can take a while, so be careful where and when you run the ANZLFITG command against a large number of files. Once the command completed, the list panel below was displayed:

```

                                Analyze Logical File Integrity
WYNDHAMW
                                                                21-05-10
14:38:08
List order . . . :   *LIB                                Position to . . .
```

```

Type options, press Enter.

    2=Move    3=Copy    4=Delete    5=Display    6=Data base relations
7=Rename
    8=Work with LF    9=Work with PF

      Logical                      Physical

Opt  File      Library    File      Library    Format
Violation
      CHECK_CSTS  QSYS2      QADBFCST  QSYS      CHECK00001
*LFLIB
      LOCATIONS  QSYS2      QADBXRDBD QSYS      LOCATIONS
*LFLIB
      REF_CST1   QSYS2      QADBFCST  QSYS      REFER00001
*LFLIB
      SCHEMATA   QSYS2      QADBIFLD  QSYS      SCHEMATA
*LFLIB
      SYCHKCST   QSYS2      QADBFCST  QSYS      SYCHKCST
*LFLIB
      SYSCOLUMNS QSYS2      QADBIFLD  QSYS      SYSCOLUMNS
*LFLIB
      SYSCOLUMNS QSYS2      QADBXSFLD QSYS      SYSCOLUMNS
*LFLIB
      SYSCST     QSYS2      QADBFCST  QSYS      SYSCST
*LFLIB

More...
Parameters or command

===>

F3=Exit      F4=Prompt    F5=Refresh   F9=Retrieve   F10=Edit
library list
F11=Display LF F12=Cancel   F17=Top      F18=Bottom

```

As usual, the various areas of the list panel and columns are explained in detail in the help text panel group accompanying this utility. Point your cursor to the location of interest and press function key F1 to display the help text for that area or column. Note that if you've installed the Work with Database Files (WRKDBF) command previously published in this column (see link to article below) option 8 and 9 will run this command. Otherwise the Work with File (WRKF) command will be executed for the list options in question.

This APIs by Example includes the following sources:

CBX215	RPGLE	Analyze Logical File Integrity - CPP
CBX215H	PNLGRP	Analyze Logical File Integrity - Help
CBX215P	PNLGRP	Analyze Logical File Integrity - Panel Group

CBX215V	RPGLE	Analyze Logical File Integrity - VCP
CBX215X	CMD	Analyze Logical File Integrity
CBX215M	CLP	Analyze Logical File Integrity - Build command

To create all the ANZLFITG command objects, compile and run the CBX215M program, following the instructions in the source header. You can also find compilation instructions in the respective source headers.

This APIs by Example article is based on a suggestion submitted by Peter Kemp in Australia. Peter also tested the ANZLFITG command and was involved in its final design. Many thanks to Peter for his help and input! If you have any ideas or suggestions for me to cover in future APIs by Example articles, please forward these to me at flensburg@novasol.dk.

IBM Documentation:

[Create Logical File \(CRTLF\) command](#)

[Create Duplicate Object \(CRTDUPOBJ\) command](#)

[Create View](#)

[Set Schema](#)

[Using DFRID to Allow Restoring Logical Files before Physical Files](#)

[Restoring logical files](#)

[Restore Deferred Objects \(RSTDFROBJ\) command](#)

[Remove Defer ID \(RMVDFRID\) command](#)

Related articles:

[APIs by Example: Working with Database Files, Fields and More](#)

[APIs by Example: Displaying and Locating a Physical File's Access Paths](#)

[APIs by Example: Print File Field Description](#)

This article demonstrates the following File APIs:

[Retrieve Database File Description \(QDBRTVFD\) API](#)

[File APIs](#)

[Database and File APIs](#)

[Retrieve the source code for this API example.](#)

Source URL: <http://iprodeveloper.com/rpg-programming/apis-example-analyzing-logical-files-using-qdbrtvfd-file-api>