

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

APIs by Example: Displaying and Locating a Physical File's Access Paths

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 09/13/2007 (All day)

System i programmers typically spend a lot of time designing, programming, and implementing the databases that constitute the core of the business applications running on the System i. Part of this effort involves the creation and maintenance of logical files, views, and indexes to provide fast and appropriate access to the data in the database, irrespective of the functions, methods, and programming languages being used to perform this access.

So whenever the need for a specific index or subset of a given physical data file arises, the next thing to do is research whether the access path required already exists, or alternatively, whether it could be built using an existing access path as base. The closest that any native i5/OS command will get you is the Display Data Base Relations (DSPDBR) command. The DSPDBR command displays a list of all the currently existing and dependent logical files, views, and indexes for any specified physical file.

This information does not, however, include any access path information, such as the dependent file's key fields, key sequence, and select/omit criteria. The idea of being able to quickly displaying all existing access path for a given physical file -- as well as optionally being able to locate any access paths, including a specified field name -- prompted me to create the Display Physical File's Access Path (DSPPFAP) command.

Access paths, whether in the form of logical files, views, or indexes, all incur the penalty of requiring an update every time the data in the underlying physical file changes. Granted, there are ways to delay this update (e.g., the Access path maintenance -- MAINT -- attribute of logical files), but at some point, before the dependent file is used and accessed again, the update must be performed to ensure that the retrieved data is correct. And access paths require auxiliary storage space to store them.

Therefore, being able to locate an existing access path meeting your requirements, rather than creating a new similar access path, makes good sense. It reduces system resource requirements and saves disk space. Even if no access path matches exactly, you might be able to arrange the creation of the new dependent file in such a way that it shares the access path with an existing file and thereby reduce the incurred maintenance costs considerably. I have included links at the end of this article to point to the IBM documentation that discusses these issues and possibilities in more detail.

The DSPPFAP command is based on two APIs, first the List Database Relations (QDBLDDBR) API which, similar to the DSPDBR command, produces a list of all dependent logical files for a given physical file. The list is written to the user space specified as the API's first parameter. If you are familiar with the use of user spaces in conjunction with APIs, you'll find the steps involved in retrieving a list of information from a user space trivial.

For the benefit of anyone not that experienced yet, here's a synopsis:

1. Create a user space calling the Create User Space (QUSCRTUS) API.
2. Call the QDBLDBR API, specifying the name and library of the user space to which the API should return information.
3. Check the API error data structure to ensure that the call completed successfully.
4. Retrieve the address of the user space by using the QUSPTRUS API, which returns a pointer to the user space. The pointer will point to the address at which the first byte of information in the user space is stored.
5. Use the returned pointer to set the address of the API user space header information data structure. To enable the setting of a data structure's address, define it using the BASED keyword, specifying the name of a pointer variable as the keyword's only parameter. The data structure will then contain the data found at the address of the pointer variable.
6. Relying on the information in the API header data structure, set the address of the list entry data structure. The first list entry is found by adding the API list offset value, given in the API user space header information data structure, to the user space pointer returned in step 3.
7. For each list entry, complete the list entry processing by adding the list entry length information returned in the API user space header information data structure. This action advances the address of the list entry data structure to the next list entry. If zero entries are returned, no list entry processing should of course be performed. You should advance the list entry address only as long as there are more list entries to retrieve.
8. Repeat step 6 for the number of list entries returned. The number of list entries is also found in the API user space header information data structure.
9. Delete the user space by using the Delete User Space (QUSDLTUS) API.

For greater detail on how to retrieve information from a user space, I recommend the articles "Getting Started with APIs, Part 2" and "APIs by Example: Retrieve Subsystem Entries API" (links to both articles are provided at the end of this article).

After I retrieve a list of all dependent logical files, the next step is to retrieve information about each file. This information should include names of the file's key fields, key sequence, select/omit specifications, and more. To get at this type of information, I use the Retrieve Database File Description (QDBRTVFD) API -- one of the more complex and challenging APIs to code. Here are some of the main considerations to take into account when coding APIs of this type:

- The QDBRTVFD API can return a huge amount of file information. Therefore, you must explicitly allocate the storage that you provide to the API as the location to return information to. This procedure lets you exceed the 65,535-byte limit of character variables and named data structures and reallocate the specific amount needed, should the initial size of the return parameter be insufficient.
- The returned information from the QDBRTVFD API is structured in such a way that, to get at the information that you are specifically interested in, you often need to define many data structures in your program. Starting from the main or header data structure of the return format that you selected when calling the API, you then retrieve the offset to the next data structure, which in turn holds the offset to yet another data structure. Jumping from one structure to the next continues until you arrive at the data structure that you are seeking.

In a previously published APIs by Example article that includes a Print File Field Description utility that also uses the QDBRTVFD API, you can find a more detailed example of the use and programming of this API (see the link at the end of this article).

After the QDBRTVFD API documentation has been deciphered, and the API call has been coded, I'm set to go. For each dependent logical file that the QDBLDBR API returns, I call the QDBRTVFD API. For each database file information data structure that the QDBRTVFD API returns, the relevant access path information is retrieved from the nested data structures. I suggest that you run the DSPPFAP CPP in the source debugger to watch the process unfold and get a closer look at how the information is retrieved.

The DSPPFAP command has the following command prompt:

```

                                Display Phys File Access Paths (DSPPFAP)

Type choices, press Enter.

File . . . . . Name

Library . . . . . *LIBL      Name, *LIBL,
*CURLIB
Include field . . . . . *ALL      Character value,
*ALL

```

Specify a physical file optionally qualified by a library to see a list of all access paths based on that file available. If you specify a value for the Include field parameter, only the access paths that have the specified field name or partial field name defined as a key field or as select/omit criteria, are included in the list. Here's an example of how such a list would look for the QADBIFLD file in library QSYS:

Display Physical File Access Paths					
WYNDHAMW					
					06-09-07
18:50:00					
File name	:	QADBIFLD		
Library	:	QSYS	Include field	. . .
File name	Library	Format	Field	Seq	Select/omit
values					
QADBIFLD	QSYS	QDBIFLD	DBILIB	A UQ	
QADBIFLD	QSYS	QDBIFLD	DBIFIL	A UQ	
QADBIFLD	QSYS	QDBIFLD	DBIFMP	A UQ	
QADBIFLD	QSYS	QDBIFLD	DBIFMT	A UQ	
QADBIFLD	QSYS	QDBIFLD	DBIPOS	A UQ	
QADBIATR	QSYS	QDBIFLD	DBIATR	A	

QADBIATR	QSYS	QDBIFLD	DBILIB	A
QADBIATR	QSYS	QDBIFLD	DBIFIL	A
QADBIATR	QSYS	QDBIFLD	DBIFMP	A
QADBIATR	QSYS	QDBIFLD	DBIFMT	A
QADBIATR	QSYS	QDBIFLD	DBIFLD	A
QADBILFI	QSYS	QDBIFLD	DBILIB	A
More...				
F3=Exit	F10=Move to top	F12=Cancel	F17=Top	
F18=Bottom				
F21=Print list	F22=Display entire select/omit			

Files whose access path does not include key fields are displayed with the Field column value set to *ARRIVAL. Files having more than one record format have each record format's group of key fields and select/omit criteria listed. Each file's or record format's group of key fields and select/omit criteria are separated by a blank line.

The Include field parameter is also available in the display panel to change or enter a selection criterion for the list function. Placing the cursor on a line in the list and pressing function key F10 moved the selected line to the top of the list, to make it easier to view a set of key fields and select/omit criteria that begin on one page and end on the next.

Function key F21 prints the currently displayed list to the job's default output queue. If the select/omit values column is not wide enough to hold the full string of values, you can place the cursor on the select/omit values column and press the F22 key to display a window with all the defined select/omit values. As always, refer to the cursor-sensitive online help text for all details about the DSPPFAP command as well as the command's display panel.

This APIs by Example includes the following sources:

```
CBX177  -- RPGLE  -- Display Physical File Access Paths - CPP

CBX177E -- RPGLE  -- Display Physical File Access Paths - UIM Exit
Program
CBX177H -- PNLGRP -- Display Physical File Access Paths - Help
CBX177P -- PNLGRP -- Display Physical File Access Paths - Panel group

CBX177V -- RPGLE  -- Display Physical File Access Paths - VCP

CBX177X -- CMD    -- Display Physical File Access Paths
CBX177M -- CLP    -- Display Physical File Access Paths - Build
command
```

To create all these objects, compile and run CBX177M. Compilation instructions are in the source headers, as usual.

Previously published related articles:

Getting Started with APIs, Part 2:

<http://www.systeminetwork.com/article.cfm?id=18648>

APIs by Example: Retrieve Subsystem Entries API:

<http://www.systeminetwork.com/article.cfm?id=53255>

APIs by Example: Print File Field Description:

<http://www.systeminetwork.com/article.cfm?id=19279>

IBM access-path-related documentation:

Describing access paths for database files:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/dbp/rbafoapath.htm>

Sharing existing access paths between logical files:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/dbp/rbafoshrap.htm>

Database Programming Manual:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/dbp/rbafo.pdf>

Display Data Base Relations (DSPDDBR) command:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/cl/dspdbr.htm>

This article demonstrates the following Database and File APIs:

List Database Relations (QDBLDDBR) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qdbldbr.htm>

Retrieve Database File Description (QDBRTVFD) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qdbrtvfd.htm>

You can retrieve the source code for this API example from:

http://www.pentontech.com/IBMContent/Documents/article/55516_329_DspPfAp.zip.

Source URL: <http://iprodeveloper.com/rpg-programming/apis-example-displaying-and-locating-physical-files-access-paths>