

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

## APIs by Example: List Open Files API, and the Display Job Open Files Command

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 03/24/2011 (All day)



IBM initially conceived APIs to provide programmers a well-documented and well-structured interface to system information that in the pre-API days was obtained by parsing CL command spooled file output or by calling IBM system internal programs directly. In addition to APIs' versatility and standardized interfaces, they also often offer much more information and details than their original CL command counterpart.

Today's APIs by Example demonstrates an API that in itself exposes more information about a job's open files than is available elsewhere. At the same time, true to the concept of a programming interface, the API approach lets you further enhance the functionality associated with the resulting Display Job Open Files (DSPJOBOPNF) CL command, compared to the corresponding native offering. The List Open Files (QDMLOPNF) API delivers the core command functionality of listing a specified job's currently open file objects.

The IBM CL commands Display Job (DSPJOB) and Work with Job (WRKJOB) both support an OPTION(\*OPNF), which, in the words of the associated help text, performs a similar service: "Files that are open for the job and the status of system and user files are shown." Programmers often refer to these commands and this option to examine and verify the files that their programs have opened, and the type of operation being performed against these files.

You can see the relative record numbers of the file records as they're being processed, and you can verify the libraries of the files opened. For anyone who has ever enjoyed the outcome of testing an update program against a production file, the latter is a very useful capability. There's a column specifying the number of I/O operations performed to the respective open files, and information about activation scope and activation groups. For many programming tasks, the DSPJOB or WRKJOB command's Display Open Files panel will help you get your job done. But in some situations, this panel has shortcomings.

There's no specification of the individual types of output being performed: write, read, write/read, and other I/O. You only see the accumulated result in a single column. The limited column size for the I/O count as well as the relative record number at some point causes overflow for jobs performing either excessive I/O for longer periods of time or processing large files. Using the function key F5 to refresh the screen after paging down one or more pages immediately takes you back to page 1. You have no way of limiting the list panel to include only particular files, libraries, types of files, or I/O.

As upcoming issues of this column will further demonstrate, you have the happy option as an API programmer, and given the presence of an appropriate API, to build the tools you need in order to make your job a little easier and the outcome of your efforts a little better. For now, since this also

presents the initial specifications for the CPP, let me show you what the DSPJOBOPNF command prompt looks like:

Display Job Open Files (DSPJOBOPNF)

Type choices, press Enter.

Job name . . . . .	*	Name, *
User . . . . .		Name
Number . . . . .		000000-999999
File name . . . . .	*ALL	Name, generic*,
*ALL		
Library . . . . .	*ALL	Name, generic*,
*ALL		
File type . . . . .	*ALL	*ALL, *BSCF,
*BSCF, *CMNF...		
		+ for more values
I/O type . . . . .	*ALL	*ALL, *ANYIO,
*READ...		
		+ for more values
Output . . . . .	*	*, *PRINT

The command's primary parameter, the job for which to list the open files, is the only one directly supported by the QDMLOPNF API. The remaining parameters enabling you to qualify which files to include in the open files list are all enforced by the CPP. You can specify a file name or a generic file name, a library name or a generic library name, and any number of file types and I/O types in order to list only a specific selection of open files. The command and all its parameters are documented in more detail in the accompanying online help text panel group.

Here's the QDMLOPNF API parameter list in its entirety:

Required Parameter Group:			
1	Receiver variable	Output	Char (*)
2	Length of receiver variable	Input	Binary (4)
3	Format of receiver information	Input	Char (8)
4	Job identification information	Input	Char (*)
5	Format of job identification info	Input	Char (8)
6	Error code	I/O	Char (*)

The first and second parameters define the program variable available for the QDMLOPNF API to return the open file information and the size of this variable, respectively. Since any arbitrary number of files may be open when the API is called, it's difficult to predict the exact amount of storage required to hold all available open file information. I therefore dynamically allocate storage for the API receiver variable. Initially, I allocate enough storage to cater for approximately 400 open files. This would cover the storage requirement in most cases. Should it not suffice, however, I repeat the API call following a reallocation of storage based on the actual amount of open file information available. This approach translates to the following piece of RPG/IV code:

```

/Free

  ApiRcvSiz = 65535;
  pOPNF0100 = %Alloc( ApiRcvSiz );

  OPNF0100.BytAvl = *Zero;

  DoU   OPNF0100.BytAvl   *Zero;

      If   OPNF0100.BytAvl > ApiRcvSiz;
        ApiRcvSiz  = OPNF0100.BytAvl;
        pOPNF0100 = %ReAlloc( pOPNF0100: ApiRcvSiz );
      EndIf;

      LstOpnF( OPNF0100
                : ApiRcvSiz
                : 'OPNF0100'
                : JIDF0100
                : 'JIDF0100'
                : ERRC0100
                );
    EndDo;

/End-Free

```

The QDMLOPNF API call is repeated until the size of the open file information is less than the size of the receiver variable (or an error condition is signaled in the API error data structure). Prior to subsequent API calls, the required amount of storage is reallocated. The dynamically allocated storage remains allocated until explicitly deallocated or the activation group in which the program runs. One method of ensuring that allocated storage is released properly irrespective of how a program ends is to run a program dynamically allocating storage in a \*NEW activation group.

This approach might, however, in some contexts constitute a bad practice due to the overhead related to creating new activation groups. So another method of protecting against storage not being released is to, for example, register a termination exit procedure. A termination exit procedure is called by the system runtime whenever a program ends due to anything other than a normal return. The registered exit procedure then is capable of releasing allocated storage, or performing any other cleanup procedure required. Note that the system value QENDJOB LMT controls the amount of time available to complete end job processing, in case job termination is the cause of the program invocation being ended.

The DSPJOBOPNF CPP therefore initially registers the TrmPgm() procedure. The TrmPgm() procedure contains all the operations that I want to be sure are run before the program for which it is

registered ends. If the CPP ends normally, the final operations performed by the program are to deregister the TrmPgm() termination exit procedure as a cleanup precaution and then execute the TrmPgm() procedure inline instead. The code snippets below outline the steps involved in performing this type of program termination control:

```

**-- Register termination exit:
D CeeRtx          Pr          ExtProc( 'CEERTX' )
D  procedure          *      ProcPtr  Const
D  token            *      Options( *Omit )
D  fb                12a     Options( *Omit )
**-- Unregister termination exit:
D CeeUtx          Pr          ExtProc( 'CEEUTX' )
D  procedure          *      ProcPtr  Const
D  fb                12a     Options( *Omit )

/Free

    CeeRtx( %Paddr( TrmPgm ): *Omit: *Omit );

...

    CeeUtx( %Paddr( TrmPgm ): *Omit );

    TrmPgm( *Null );

/End-Free

**-- Terminate program:
P TrmPgm          B
D                  Pi
D  pPtr            *      Const

/Free

    CloApp( UIM.AppHdl: CLO_NORM: ERRRC0100 );

    DeAlloc(n)  pOPNF0100;

    *InLr = *On;
    Return;

/End-Free

P TrmPgm          E

```

The QDMLOPNF API's third parameter specifies the format in which you want the API to return the open file information. Currently only a single format, OPNF0100, is offered. A similar limited range of options exists for the fourth parameter, the *Job identification information* pointing the API to the job for which to produce the open file listing. Again, a single format is available, the JIDF0100 format, the name of which must be specified as the fifth parameter when you call the QDMLOPNF API. Here's the layout of the JIDF0100 parameter structure using an offset base of 1:

Offset	Field	Data type
1	Job name	Char(10)
11	User name	Char(10)
21	Job number	Char(6)
27	Internal job identifier	Char(16)
43	Reserved	Char(2)
45	Thread indicator	Binary(4)
49	Thread identifier	Char(8)

The JIDF0100 format is used by a number of work management APIs to let you identify the scope of the job information to return right down to individual thread level. You identify the job by job name, user name, and job number, or by the internal job identifier. The latter is a system internal identifier of any given job that is returned by other APIs in order to allow subsequent API calls to locate the job faster than possible with the qualified job name. That's all straightforward. Getting the thread indicator right, however, requires a closer look at the description of this parameter:

#### **Thread indicator**

The value that is used to specify the thread within the job for which information is to be retrieved.

The following values are supported:

0 The value in the thread identifier field should be used to locate the thread.

1 Information should be retrieved for the thread in which this program is running.

The combination of the internal job identifier, job name, job number, and user name fields also must identify the job containing the current thread.

2 Information should be retrieved for the initial thread of the identified job.

3 Information should be retrieved for all threads within the specified job.

Specifying a zero for the thread indicator parameter causes the QDMLOPNF API to return open file information only for the thread identified by the thread identifier parameter. Specifying the value one retrieves information only for the job calling the QDMLOPNF API. Entering job identification values identifying another job than the current one causes the API call to fail. Values two and three both support current as well as other jobs, but the value two only returns information for the specified job's initial thread. In this case, I want to see all open files associated with any given job, so I specify the value three for the thread indicator parameter.

As for the sixth and final API parameter, the API error data structure format ERRC0100, this has been demonstrated and discussed to great extent in other, earlier articles. I've included links to articles discussing the concept of the API error data structure as well as dynamic memory allocation at the end of this article, in case you want to read up on the details and specifics. In the same section,

you'll also find a number of links to IBM documentation explaining some of the other concepts discussed or involved in today's article or code.

Now, on to the outcome of our efforts so far. The Display Open Files panel presented by the DSPJOBOPNF command mainly differs from the original version in that the primary list view panel showing the open file I/O information has been divided into two panels. The initial panel displayed identifies the open file and includes information about the file type, member/device name, and relative record number:

Display Open Files								
WYNDHAMW					11-03-11			
15:48:40								
Job: QPADEV0007		User: CARSTEN		Number: 966052				
Open data paths . . . . . : 4								
Relative		Member/		Record		File ---Open---		
File	Library	Device	Format	Type	Opt	Shr	Nbr	
Record								
QSN132	QSYS	CF101HOA	USRRCD	DSP	IO	NO		
QDUODSPF	QPDA	CF101HOA	MSGSFC	DSP	IO	NO		
QDUI132	QSYS	CF101HOA	USRRCD	DSP	IO	NO		
QAOKL02A	QUSRSYS	QAOKL02A	WOSFMT01	LGL	I	YES	1	
60								
Bottom								
Press Enter to continue.								
F3=Exit F5=Refresh F11=Display I/O details F12=Cancel								
F24=More keys								

The function keys let you toggle the list views, execute the Work with Job (WRKJOB) command, and position the open files list to top and bottom, respectively. Function key F10 lets you move the list record selected with the cursor to the top of the panel. Pressing function key F5 maintains the list's current position based on the top file's ordinal number in the list. This implies that if files preceding the current top file have been opened or closed since the list was last built, the top file may consequently change. Under most circumstances, however, the top file remains the same following a list refresh.

The second open files list view contains the detailed open file I/O information:

```

                                Display Open Files

WYNDHAMW                                11-03-11

15:56:56
Job:   QPADEV0007      User:   CARSTEN      Number:   966052

Open data paths . . . . . :    4

-----I/O Count-----
-----
File      Library      Read      Write      Write/Read
Other I/O
QSN132    QSYS              0          0          4
  1
QDUODSPF  QPDA             73         409         1
  1
QDUI132   QSYS              0          0         12
  1
QAOKL02A  QUSRSYS           1          0          0
  0

Bottom
Press Enter to continue.

F10=Move to top  F16=Job menu  F17=Top    F18=Bottom  F24=More
keys

```

As the above open files list panel example demonstrates, for each open file listed, the following I/O event types are counted individually:

**Read**        The number of successful read operations. If record blocking is not in effect for the file, this is the number of records. If record blocking is in effect for the file, this is the number of record blocks. A read in this context defines the transfer of a record or a block of records from a file to a program. The data is made available to the program once the read has been successfully completed.

**Write**        The number of successful write operations. If record blocking is not in effect for the file, this is the number of records. If record blocking is in effect for the file, this is the number of record blocks. A write in this context defines the transfer of a record or a block of records from a program to a file.

**Write/Read**    The number of successful write/read operations. A write/read in this context defines the combination of write and read as one single operation. An example of a combined write/read operation is a write performed to a display file format, which then immediately after the completed write operation waits for an input operation being performed to the same display file format.

**Other I/O**        The number of successful I/O operations of the following types:

- o update
- o delete
- o change end-of-data
- o force end-of-data
- o force end-of-volume
- o release record lock



```
o acquire/release program device
```

As I mentioned earlier, you can use the function key F5 to refresh the screen and thereby the I/O count. The DSPJOBOPNF command's third list view panel essentially displays the same activation group information as the native version, so I don't go into more details here. All panels as well as the list columns are further explained in the cursor-sensitive help text included with the DSPJOBOPNF command.

**This APIs by Example includes the following sources:**

```
CBX227  -- RPGLE  -- Display Job Open Files - CPP
CBX227E -- RPGLE  -- Display Job Open Files - UIM Exit Program
CBX227H -- PNLGRP -- Display Job Open Files - Help
CBX227P -- PNLGRP -- Display Job Open Files - Panel Group
CBX227X -- CMD    -- Display Job Open Files

CBX227M -- CLP    -- Display Job Open Files - Build command
```

To create all these objects, compile and run the CBX227M program, following the instructions in the source header. You'll also find compilation instructions in the respective source headers.

**Related Articles:**

[A Beginner's Guide to APIs \(API Error Data Structure\)](#)

[Introduction to Pointers in RPG \(Dynamic Memory Allocation\)](#)

**IBM Documentation:**

[Work Management Job Concepts - Jobs](#)

[Threads](#)

[Memory Management Operations \(RPG/IV\)](#)

[Managing the Default Heap Using RPG Operations](#)

[Jobs system values: Maximum time for immediate end](#)

[Data Management Operations Overview](#)

[Data Management Manual \(PDF\)](#)

**This article demonstrates the following APIs:**

[List Open Files \(QDMLOPNF\) API](#)

[Register Call Stack Entry Termination User Exit Procedure \(CEERTX\) API](#)

[Unregister Call Stack Entry Termination User Exit Procedure \(CEEUTX\) API](#)

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

**Source URL:** <http://iprodeveloper.com/rpg-programming/apis-example-list-open-files-api-and-display-job-open-files-command>