

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

APIs by Example: Directing API Output to Output Files Using the SQL CLI APIs

[*System iNetwork Programming Tips Newsletter*](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 02/25/2010 (All day)

Many IBM CL commands provide an output file option that enables the command to direct its output to a database file for further processing or collection of data over time. Due to the fact that practically no APIs offer this option, I once in a while I get requests from readers asking me how to provide for an output file facility to a specific API. This APIs by Example column demonstrates one of the methods available to accommodate the API output file requirement, and it involves the use of the SQL Call Level Interface (CLI) APIs.

The example I present uses the List Network Connection (QtocLstNetCnn) API as the source of information to be stored in an output file, but essentially any list API could have provided the foundation. When I wrote the List Network Connection (LSTNETCNN) command, it was therefore my intention that the code could easily be copied and adapted to serve as the starting point for other similar list API-based output file utilities.

The native CL commands supporting output files typically use model files located in either the QSYS or the QUSRSYS system libraries. At the point where the command is ready to produce its output, and if the target file does not already exist, the model file is simply copied to the location specified, and the output records are added. To avoid the dependency on model files and to keep the file creation process within the control of the program producing the output going into the file, I decided to use the SQL CLI APIs for that purpose.

The SQL CLI APIs enable you, among many other things, to both create and insert records into a database file using SQL statements executed directly from within your program, and because they're included with the operating system at no charge, the SQL CLI APIs are readily available on any IBM i system. While RPG/IV embedded SQL from an SQL perspective provides you with many of the same options as the SQL CLI APIs, an embedded SQL approach requires the SQL Development Kit, product 5722ST1 at release 5.4, to be licensed and installed, so using the SQL CLI APIs for the purpose at hand eliminates those concerns.

The SQL CLI API topic as such has been covered to an impressive extent in a number of articles written by Scott Klement, the editor of this newsletter, so I've kept that part out of scope of this article and included links to a list of previously published SQL CLI articles below. I will of course go through the specific use of the SQL CLI APIs in the code accompanying this article, but for a broader introduction to this topic, I recommend you look up the mentioned articles. Also note that the LSTNETCNN CPP takes advantage of the SQLCLI_H copy member created by Scott Klement and included with the listed articles, so be sure to follow the instructions at the end of this article in order to download this copy member before attempting the creation of the LSTNETCNN command objects.

The QtocLstNetCnn API offers two almost identical return formats, one for Internet Protocol version 4 (IPv4) connections and another for Internet Protocol version 6 (IPv6) connections. The task of

combining the two formats into a single output file format capable of holding the API output, irrespective of IP version, is not all that difficult. Here's a list of the information available:

- Remote IP address
- Local IP address
- Remote port
- Local port
- TCP state
- Idle time
- Bytes in
- Bytes out
- Connection open type
- Net connection type
- Line description (IPv6 only)

I add the following fields to complete the output file format:

- IP version (4=IPv4, 6=IPv6)
- TCP state code (LSTN, SYN, SYN, EST, FIN₁, etc.)

Only a few SQL statements are required to create and label the output file and its fields appropriately. This is naturally the part of the CPP that will need to be adapted in order to support a different output file format. Once the SQL CLI environment has been initialized and a connection has been made to the local database, the following statement will take care of creating a file by the name and library location specified on input to the program and stored in qualified format in the *SQLTable* variable:

```
SQLStmt = 'CREATE TABLE ' + SQLTable + ' ('
        'IPVERS CHAR      (1)      NOT NULL WITH DEFAULT, ' +
        'RMTADR CHAR   (45)      NOT NULL WITH DEFAULT, ' +
        'LOCADR CHAR   (45)      NOT NULL WITH DEFAULT, ' +
        'RMTprt NUMERIC  (5,0) NOT NULL WITH DEFAULT, ' +
        'LOCPRT NUMERIC  (5,0) NOT NULL WITH DEFAULT, ' +
        'TCPSTT NUMERIC  (2,0) NOT NULL WITH DEFAULT, ' +
        'IDLTIM NUMERIC (10,0) NOT NULL WITH DEFAULT, ' +
        'BYTIN  NUMERIC (20,0) NOT NULL WITH DEFAULT, ' +
        'BYTOUT NUMERIC (20,0) NOT NULL WITH DEFAULT, ' +
        'CNNOPT NUMERIC  (1,0) NOT NULL WITH DEFAULT, ' +
        'NETCNT CHAR   (10)      NOT NULL WITH DEFAULT, ' +
        'JOBUSR CHAR   (10)      NOT NULL WITH DEFAULT, ' +
        'LINDSC CHAR   (10)      NOT NULL WITH DEFAULT, ' +
        'TCPSTC CHAR   (10)      NOT NULL WITH DEFAULT'
        ') RCDfmt NETCNr';

rc = SQLExecDirect( stmt: SQLStmt: SQL_NTS );
```

The `SQLExecDirect()` API executes the SQL statement stored in the *SQLStmt* variable immediately. In order to provide a descriptive text for the newly created file as well as column headings for the file's fields, the following two SQL statements and their subsequent execution will do the job:

```
SQLStmt = 'LABEL ON TABLE ' + SQLTable + ' IS '
        'TCP/IP network connections list';
```

```

rc = SQLExecDirect( stmt: SQLStmt: SQL_NTS );

SQLStmt = 'LABEL ON COLUMN ' + SQLTable + ' ' +
          '(IPVERS IS 'TCP/IP version', ' +
          'RMTADR IS 'Remote address', ' +
          'LOCADR IS 'Local address', ' +
          'RMTprt IS 'Remote port', ' +
          'LOCPRT IS 'Local port', ' +
          'TCPSTT IS 'TCP state', ' +
          'IDLTIM IS 'Idle time', ' +
          'BYTIN IS 'Bytes in', ' +
          'BYTOUT IS 'Bytes out', ' +
          'CNNOPT IS 'Conn. open type', ' +
          'NETCNT IS 'Net conn. type', ' +
          'JOBUSR IS 'Assoc. user profile', ' +
          'LINDSC IS 'Line description', ' +
          'TCPSTC IS 'TCP state code' )';

rc = SQLExecDirect( stmt: SQLStmt: SQL_NTS );

```

To also include a text attribute for each field, I execute the statement below:

```

SQLStmt = 'LABEL ON COLUMN ' + SQLTable + ' ' +
          '(IPVERS TEXT IS '4=IPv4, 6=IPv6', ' +
          'RMTADR TEXT IS 'Formatted address', ' +
          'LOCADR TEXT IS 'Formatted address', ' +
          'RMTprt TEXT IS '0-65535', ' +
          'LOCPRT TEXT IS '1-65535', ' +
          'TCPSTT TEXT IS '0-11', ' +
          'IDLTIM TEXT IS 'Milliseconds (ms)', ' +
          'BYTIN TEXT IS 'Byte count', ' +
          'BYTOUT TEXT IS 'Byte count', ' +
          'CNNOPT TEXT IS '0=Pas, 1=Act, 2=n/s', ' +
          'NETCNT TEXT IS '*TCP, *UDP, *IPS', ' +
          'JOBUSR TEXT IS 'Profile name', ' +
          'LINDSC TEXT IS 'Object name', ' +
          'TCPSTC TEXT IS 'State abbrev.' )';

rc = SQLExecDirect( stmt: SQLStmt: SQL_NTS );

```

None of the three LABEL ON statements are of course mandatory in terms of being able to store the API output, but spending the little extra effort will make it much easier and comprehensible to work with the file and its content later. The next step is to prepare the actual insert of the output file records. The SQL insert statement follows common SQL syntax rules, naming the file name to insert records into as well as the fields targeted by the operation:

```

SQLStmt = 'INSERT INTO ' + SQLTable + ' ' +
          '(IPVERS, RMTADR, LOCADR, RMTprt, LOCPRT, TCPSTT, ' +
          ' IDLTIM, BYTIN, BYTOUT, CNNOPT, NETCNT, JOBUSR, ' +
          ' LINDSC, TCPSTC) ' +
          'VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)';

```

The VALUES keyword, however, specifies only question marks in place of the actual values. The question marks are called *parameter markers* in this context. Each parameter marker designates a value to be specified at statement execution time. How that is done, I'll show you in a moment, for now I will simply have the SQLPrepare() API process the above statement:

```
rc = SQLPrepare( stmt: SQLStmt: SQL_NTS );
```

The SQLPrepare() API associates the SQL statement specified in the second parameter with the input statement handle specified as the first parameter. Following the prepare process, other SQL CLI APIs have access to the prepared SQL statement when the statement handle is specified as input to these APIs. All that remains now is to tie each parameter marker to the location where the value to replace the parameter marker at SQL statement execution time is found. To do so I create a data structure capable of holding all the QtocLstNetCnn API-supplied network connection information that should go into the output file:

```

**-- SQL insert values:
D SQLValue          Ds              Qualified
D IpVers              1a
D RmtAdr              45a
D LocAdr              45a
D RmtPrt              5s 0
D LocPrt              5s 0
D TcpStt              5s 0
D IdlTim              10s 0
D BytIn               20s 0
D BytOut              20s 0
D CnnOpt              1s 0
D NetCnt              10a
D JobUsr              10a
D LinDsc              10a
D TcpStc              5a

```

For each parameter marker and in the same order as specified on the SQL INSERT INTO statement previously prepared, I then call the SQLBindParameter() API, as in the following example, binding the first parameter marker to the *SQLValue.IpVers* variable and likewise with the remaining variables:

```

rc = SQLBindParameter( stmt
                      : 1
                      : SQL_PARAM_INPUT
                      : SQL_CHAR
                      : SQL_CHAR
                      : %Size( SQLValue.IpVers )
                      : *Zero
                      : %Addr( SQLValue.IpVers )
                      : SQL_IGNORE_INT
                      : lenParm1
                      );

...

rc = SQLBindParameter( stmt
                      : 4

```

```

        : SQL_PARAM_INPUT
        : SQL_NUMERIC
        : SQL_NUMERIC
        : %Len( SQLValue.RmtPrt )
        : %DecPos( SQLValue.RmtPrt )
        : %Addr( SQLValue.RmtPrt )
        : SQL_IGN_INT
        : *Omit
    );

    ...

```

Here's a brief explanation of each of the SQLBindParameter() API parameters, as they apply to the context given here:

1. The statement handle identifying the previously prepared SQL statement.
2. Parameter marker number, ordered sequentially left to right, starting at 1.
3. The type of parameter. Primarily relevant for stored procedure calls. The SQL_PARAM_INPUT constant is used here.
4. The C programming language data type of the parameter. See API documentation for more information.
5. The SQL data type of the parameter. Same value as specified for parameter 4.
6. The precision or maximum length of the corresponding parameter marker.
7. Scale of the corresponding parameter if packed or zoned (SQL_DECIMAL or SQL_NUMERIC). Also used to specify timestamp sub-second precision, otherwise ignored.
8. Parameter value pointer. The address of the program variable containing the parameter value at SQL statement execution time.
9. Not used.
10. The program integer variable containing the length of the character variable specified for parameter 8. Note that the length is taken at SQL statement execution time, so it is important that the variable specified for this parameter is an exact match to the API prototype. This to ensure that the compiler does not assign a temporary variable, which could possibly contain an unexpected value at execution time.

In this example, I've used zoned numeric variables, but if you, for example, prefer packed decimal, simply change the relevant SQLValue source variable's data type to p(acked), parameters 4 and 5 to SQL_DECIMAL, and the SQL CREATE TABLE statement accordingly. Here's an adapted version of the code demonstrating that:

```

**-- SQL insert values:
D SQLValue          Ds                      Qualified
D  IpVers            1a
D  RmtAdr            45a
D  LocAdr            45a
D  RmtPrt            5p 0 ...

SQLStmnt = 'CREATE TABLE ' + SQLTable + ' ('
          'IPVERS CHAR      (1)      NOT NULL WITH DEFAULT, ' +
          'RMTADR CHAR      (45)     NOT NULL WITH DEFAULT, ' +
          'LOCADR CHAR      (45)     NOT NULL WITH DEFAULT, ' +
          'RMTprt DECIMAL   (5,0) NOT NULL WITH DEFAULT, ' ...

rc = SQLBindParameter( stmt

```

```

: 4
: SQL_PARAM_INPUT
: SQL_DECIMAL
: SQL_DECIMAL
: %Len( SQLValue.RmtPrt )
: %DecPos( SQLValue.RmtPrt )
: %Addr( SQLValue.RmtPrt )
: SQL_IGN_INT
: *Omit
);
...

```

Note that in the SQL CLI API code examples accompanying the SQL CLI articles previously referred to, the SQLBindParam() API is usually employed to establish the correlation between SQL statement parameter marker and corresponding program variables. The SQLBindParameter() API is a later and enhanced version of the original SQLBindParam() API. Both APIs are still supported by IBM, but the former is now recommended by IBM to be used in new code, as support of the latter might be withdrawn at some point in the future.

At this point everything is ready for the network connection information to be inserted into the output file. So I call the QtocLstNetCnn API and process all the data returned by the API to the user space specified on the API call. This follows the normal conventions applying to processing output from list APIs, which return information by means of user spaces. If you're interested in more details on this subject, I've included a link below to an article offering a thorough explanation. Finally, for each API list record extracted from the user space, I load the information from the API output structure to the corresponding fields in the SQLValue data structure, and then I run the SQLExecute() API to execute the SQL INTO statement prepared earlier:

```

If  PxIpVers = '4';
  SQLValue.IpVers = '4';
  SQLValue.RmtAdr = NCNN0100.RmtAdr;
  SQLValue.LocAdr = NCNN0100.LocAdr;
  SQLValue.RmtPrt = NCNN0100.RmtPort;
  ...
  SQLValue.JobUsr = NCNN0100.AscUsrPrf;
  SQLValue.LinDsc = *Blanks;
  SQLValue.TcpStc = GetTcpStt( NCNN0100.TcpState );
Else;
  SQLValue.IpVers = '6';
  SQLValue.RmtAdr = NCNN0200.RmtAdr;
  SQLValue.LocAdr = NCNN0200.LocAdr;
  SQLValue.RmtPrt = NCNN0200.RmtPort;
  ...
  SQLValue.JobUsr = NCNN0200.AscUsrPrf;
  SQLValue.LinDsc = NCNN0200.LinDsc;
  SQLValue.TcpStc = GetTcpStt( NCNN0200.TcpState );
EndIf;

rc = SQLExecute( stmt );

```

This will insert one record into the output file, and I'll then simply repeat the above process until all API output records have been inserted. As you will see if you take a closer look at the LSTNETCNN

CPP, there are some SQL CLI environment initialization and cleanup steps involved also, but in order to reuse the code presented here for other list APIs, the steps discussed above are the ones you will need to adapt in order to make output file part work. As for the SQL CLI approach in its entirety, you'll find all the details explained in the articles located at the links below, as for example "Retrieve an SQL Result Set with RPG."

Now back to the LSTNETCNN command. I've included the command prompt displaying all the command's parameters below, although normally only the appropriate (IPv4 or IPv6) address range parameters are displayed, depending on the input in the IP version parameter:

```

List Network Connections (LSTNETCNN)

Type choices, press Enter.

Output file . . . . . Name
Library . . . . . *LIBL Name, *LIBL,
*CURLIB
Replace or add records . . . . . *ADD *REPLACE, *ADD
IP version . . . . . *IPV4 *IPV4, *IPV6

Local IPv4 address range:

Lower value . . . . . *
Upper value . . . . . *ONLY

Local IPv6 address range:

Lower value . . . . . *
Upper value . . . . . *ONLY

Local port range:

Lower value . . . . . * 1-65535, *
Upper value . . . . . *ONLY 1-65535, *ONLY

Remote IPv4 address range:

Lower value . . . . . *
```

```

Upper value . . . . . *ONLY

Remote IPv6 address range:

Lower value . . . . . *

Upper value . . . . . *ONLY

Remote port range:

Lower value . . . . . *          1-65535, *
Upper value . . . . . *ONLY      1-65535, *ONLY

```

You'll note that in contrast to the IBM command output file convention, there's no option of specifying a file member. IBM commands create output files with a MAXMBRS(*NOMAX) attribute and allow you to specify a member name, in turn providing an option to add more members to the same output file. Since SQL has no notion of file members and files created by the SQL CREATE TABLE statement consequently allow only one member, the SQL CLI API approach chosen for the LSTNETCNN command effectively eliminates multi-member support for this command.

Any attempt to change an SQL table's MAXMBRS attribute to a value exceeding 1 is honored with the diagnostic message CPD3213 *Maximum-member value not valid for file &1* followed by escape message CPF7304.

The command and its parameters are documented in full detail in the online help text panel group, but here are some additional comments: You specify the library qualified name of the output file. If it does not exist, it will be created using the SQL CLI API calls described above. If the file already exists, you have the option of specifying whether the generated output should be added to the records already found in the output file, if any, or whether the current record content should be replaced. In the latter case, the output file is cleared prior to running the list request, irrespective of records being found or not.

The QtocLstNetCnn API supports a connection list qualifier parameter for each of the two Internet Protocol versions. The selection parameters included in the API list qualifier data structure are exposed by the LSTNETCNN command's IP address and port range selection parameters. You have the option of limiting the network connection list output to a specified local or remote IP address range and/or a local or remote port range.

The LSTNETCNN command uses the inet_pton() Sockets Network API to validate any IP addresses specified for the IPv4 or IPv6 address ranges. The primary purpose of this API is to convert an IPv4 or IPv6 address in its standard text presentation form into its numeric binary form, a capacity also employed by the LSTNETCNN command in setting up the API list qualifier parameter, which depends on the binary IP address format. But the inet_pton() API is also useful in terms of IP address validation as its return value indicates any of the three following possible outcomes of the conversion operation:


```

1 = Conversion operation was successful
0 = Conversion operation was not successful: Input format is not a
    valid IPv4
    or IPv6 address string
-1 = Conversion operation was not successful: Call error detected by
    API

```

A return value of -1 indicates the API call itself ended in error, while a return value of zero signals that the format of the specified IP address was not valid. So once you get the API call working, you'll either get a zero or a 1 back from the API, depending on whether the specified IP address is valid. The LSTNETCNN command also verifies the record format level identifier of a specified existing output file to ensure that no attempts are made to direct output to a file with an invalid record format.

This verification is based on the List Record Format (QUSLRCD) API and the valid record format identifier stored in the CPP. The latter is obtained using the Display File Description (DSPFD) command following the first successful execution of the LSTNETCNN command and subsequently entered into the CPP's FMT_LVLID global constant, upon which the program is recompiled. The valid format level identifier for the LSTNETCNN command specifies the following value:

```
D FMT_LVLID      C      '30E055F0F9848'
```

In case you change the LSTNETCNN output file format or use the CPP as a starting point for you own list commands, you'll consequently need to update the FMT_LVLID constant to reflect the change accordingly. Also please consider that I've added code to take care of possible messages being sent from the SQL CLI APIs to your job's job log. Especially if you run the command in debug mode, you'll see a lot of SQL CLI messages being sent to the job log. This is in accordance with the way the SQL runtime in general behaves when detecting debug mode being active. If you decide to preserve this behavior, you'll want to eliminate the RmvLogLst() function call in the CPP.

This APIs by Example includes the following sources:

```

CBX212  -- RPGLE  -- List Network Connections - CPP
CBX212H -- PNLGRP -- List Network Connections - Help
CBX212V -- RPGLE  -- List Network Connections - VCP
CBX212X -- CMD    -- List Network Connections

CBX212M -- CLP    -- List Network Connections - Build command

```

To create all these List Network Connection command objects, compile and run the CBX212M program, following the instructions in the source header. As always, the compilation instructions are also included in the respective source headers.

For the LSTNETCNN command processing program CBX212 to compile, you'll need to download and copy the SQLCLI_H member mentioned earlier to a QRPGLSRC source file in your job's library list. At the very end of this article, I've provided a link to a zip file containing the correct version of the SQLCLI_H copy member. In addition to the SQLCLI_H member, I've also in my code included and adapted the generic SQL CLI API Check_error() function introduced by Scott Klement in his SQL CLI API article series. Many thanks to Scott!

SQL CLI API related articles:

[Read and Write LOBs from an RPG Pointer Field](#)

[Retrieve an SQL Result Set from a Stored Procedure with Parameters](#)

[Fetch Multiple Records with SQL CLI](#)

[Websites with SQL CLI Information](#)

[Retrieve an SQL Result Set with RPG](#)

[Database Access from CL with SQL CLI](#)

[Efficient Character Processing with CLI](#)

IBM SQL CLI Documentation:

[DB2 UDB CLI functions - 5.4](#)

[Call Level Interface \(CLI\) APIs - 5.4](#)

[SQL call level interface - 5.4](#)

[Data types and data conversion in DB2 for i5/OS CLI functions](#)

[Determining equivalent SQL and ILE RPG data types](#)

[Differences between DB2 UDB CLI and embedded SQL](#)

[SQL CLI Frequently Asked Questions](#)

[SQL CLI What's new for V6R1](#)

[Technical document: OS/400 and i5/OS SQL CLI \(ODBC\) Documentation and FAQs](#)

[DB2 for i Tips & Technical Papers - SQL CLI](#)

[Technical reference: CLI Programs in RPG](#)

User Space List API article:

[APIs by Example: Retrieve Subsystem Entries API](#)

This article demonstrates the following TCP/IP Management & Socket Network APIs:

[The List Network Connections \(QtocLstNetCnn\) API](#)

[Convert IPv4 and IPv6 Addresses Between Text and Binary Form Function](#)

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

[The prerequisite SQLCLI H copy member is available as part 56657 600 CliClob.zip.](#)

Source URL: <http://iprodeveloper.com/rpg-programming/apis-example-directing-api-output-output-files-using-sql-cli-apis>