

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

APIs by Example: Zip and Unzip Files with the New 7.1 Zip API Support

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 12/08/2011 - 2:00am

[In a recent IBM announcement, IBM revealed that zip and unzip file support had been developed and PTF'd for release 7.1.](#) This support comes in the form of two ILE APIs, QzipZip and QzipUnzip, respectively. At the end of this article, I've included information about the 7.1 PTFs delivering the zip support APIs. One PTF installs the QZIPUTIL service program containing the aforementioned APIs, and another PTF copies the associated header files to the QSYSINC library. The zip support was part of a major refresh of IBM i 7.1, and I suggest you follow the above link to familiarize yourself with all the details, which might include other enhancements of interest.

After a quick study of the zip APIs' documentation and header files, I knew that it would be quite useful to create a couple of CL command interfaces to make the zip and unzip services immediately available, wherever and whenever the common requirement of zipping or unzipping a file or directory on IBM i was encountered. I therefore decided to write the Zip File (ZIPF) and Unzip File (UNZIPF) CL commands. The CPPs also offer RPG/IV examples of how to code the two corresponding APIs, should you want to integrate zip or unzip functionality directly in your programs. Today's APIs by Example brings you the details.

The IBM announcement says that the QzipZip and QzipUnzip APIs are available with the most recent *IBM HTTP SERVER FOR I* group PTF, which at the time of writing amounts to level 10. However, it quickly became apparent that this is not the case. I expect the APIs will be included in the next update of the HTTP group PTF, although I do not know that for sure, so we'll have to wait and see. Doing a search on IBM's APAR and PTF database, however, allowed me to identify the two PTFs including the APIs as well as the associated QSYSINC library header files, respectively, and as mentioned you'll find links to the PTF cover letters below.

Although the QzipZip and QzipUnzip APIs were just recently released, the IBM i 7.1 Information Center's API section already includes the API documentation for these APIs. Surprisingly however, the online QzipZip and QzipUnzip API documentation specifies only the APIs parameter lists in C notation, as the following excerpt from the API manual's UNIX-Type API section shows:

```
Compress Files and Directories (QzipZip) API

#include

void QzipZip(
    Qlg_Path_Name_T * fileToCompress,
    Qlg_Path_Name_T * compressedFileName,
    char * formatName,
    char * zipOptions,
    char * errorStruct) ``
```

Decompress an archive file (QzipUnzip) API

```
#include
void QzipUnzip(
    Qlg_Path_Name_T * compressedFileName,
    Qlg_Path_Name_T * dirToPlaceDecompFiles,
    char * formatName,
    char * unzipOptions,
    char * errorStruct)
```

As it turned out, the new QZIPUTIL RPG/IV header file in library QSYSINC actually also includes the RPG/IV prototypes for the two APIs, so the missing parameter list definition is not critical for anyone unfamiliar with C. Should you at some point be challenged with deciphering a C prototype for which IBM did not do the job for you, you will, however, find plenty of help in the document *Converting from C prototypes to RPG prototypes*, written by Barbara Morris of IBM and published on Scott Klement's website. A link to the document is included below.

Anyway, if you're more comfortable with IBM's usual API parameter list notation, here's my take on how the corresponding Zip and Unzip API documentation would look in the IBM Information Center API manual, given the above C prototypes:

Compress Files and Directories (QzipZip) API

Required Parameter Group:

1	File to zip	Input	Char(*)
2	Zip file name	Input	Char(*)
3	Zip Options format name	Input	Char(8)
4	Zip options	Input	Char(*)
5	Error code	I/O	Char(*)

Decompress an archive file (QzipUnzip) API

Required Parameter Group:

1	Zip file name	Input	Char(*)
2	Unzip to directory	Input	Char(*)
3	Unzip options format name	Input	Char(8)
4	Unzip options	Input	Char(*)
5	Error code	I/O	Char(*)

In the following paragraphs, I briefly walk you through the parameters for both APIs, which are relatively few in number and pretty straightforward. Both APIs' first and second parameter is a Qlg_Path_Name_t structure, the first one pointing to the object to be processed, and the second one pointing to where the outcome of the process should be placed. In addition to allowing you to specify a path name, the Qlg_Path_Name_t structure also provides for a set of parameters defining all relevant information about how the receiving API should interpret the path name string in order to arrive at the correct path name:

Qlg_Path_Name_t structure

Offset		Type	RPG/IV	Field
Dec	Hex			
0	0	BINARY(4)	10i 0	CCSID
4	4	CHAR(2)	2a	Country or region ID
6	6	CHAR(3)	3a	Language ID
9	9	CHAR(3)	3a	Reserved
12	C	BINARY(4)	10i 0	Path type indicator
16	10	BINARY(4)	10i 0	Length of path name
20	14	CHAR(2)	2a	Path name delimiter character
22	16	CHAR(10)	10a	Reserved
32	26	CHAR(*)	5000a	Path name (or pointer to path name)

You must specify a Coded Character Set Identifier (CCSID), a country or region ID, a language ID, as well as the path type being either a character string or a pointer to a character string, the length of path name, and the path name delimiter character. All this information is used by the API in question to ensure that the specified path name is addressed correctly. Luckily, most of the parameters in the `Qlg_Path_Name_t` structure take a default value pointing to the corresponding job attribute currently in effect. As for the path name delimiter character, note that the Zip APIs accept only a forward slash (/).

The following data structure definition shows how the above `Qlg_Path_Name_t` specification is translated into RPG/IV. The aforementioned default values are specified for all the parameters supporting this feature:

```

**-- Global constants:
D CUR_CCSID          c              0
D CUR_CTRID          c             x'0000'
D CUR_LNGID          c             x'000000'
D CHR_DLM1           c              0
**-- Qlg_Path_Name_t API path:
D Qlg_Path_Name_t...
D                      Ds              Qualified Align
D  CcsId                10i 0 Inz( CUR_CCSID )
D  CtrId                2a   Inz( CUR_CTRID )
D  LngId                3a   Inz( CUR_LNGID )
D                      3a   Inz( *Allx'00' )
D  PthTypI              10i 0 Inz( CHR_DLM1 )
D  PthNamLen            10i 0
D  PthNamDlm            2a   Inz( '/' )
D                      10a   Inz( *Allx'00' )
D  PthNam                1024a
D  pPthNam              *   Overlay( PthNam )

```

In a previous article, I discussed the `Qlg_Path_Name_t` structure in more detail, and I include a link to this article below. As for the Zip and Unzip APIs' parameter lists in particular, the zip operation expects you to employ the `Qlg_Path_Name_t` structure to specify a path to the file or directory that you want to zip as the first parameter, and the name of the zip file archive to store the zipped object(s) in as the second parameter. Likewise, for the unzip operation, you specify the zip file name to unzip as the first parameter, and the directory in which you want the unzipped object(s) to be placed as the second parameter.

Both APIs also support a number of options to apply for the zip operation being performed. These options are passed in another structure, whose format name must be specified as API parameter number three, and the actual option structure as parameter four. The Zip API option format ZIP00100 has the following definition:

Zip options structure ZIP00100				
Offset		Type	RPG/IV	Field
Dec	Hex			
0	0	CHAR(10)	10a	Verbose option
10	A	CHAR(6)	6a	Subtree option
16	10	CHAR(512)	512a	Comment
528	210	BINARY(4)	10u 0	Length of the comment
		UNSIGNED		

The *Verbose option* specifies whether verbose messages are to be printed to the standard out during the compression process. The system itself does not set up stdin, stdout, stderr descriptors, and it is the responsibility of the user of this API to set the descriptors when using this option.

The *Subtree option* specifies whether directory subtrees are included or not when creating an archive file. And the *Comment* option allows you to add a comment in the job CCSID to the newly created archive file. The corresponding unzip options structure UNZIP100 should be defined as follows:

Unzip options structure UNZIP100				
Offset		Type	RPG/IV	Field
Dec	Hex			
0	0	CHAR(10)	10a	Verbose option
10	A	CHAR(6)	6a	Replace option

The *Replace option* specifies whether an existing file needs to be replaced or not if a file by the same name already exists in the target path. This option applies only to file objects; directory names are ignored. The verbose option is also supported for the unzip operation. As noted above, the verbose option relies on a programming effort provided by the caller of the API. I've included a link below to an article written by Scott Klement discussing the setup involved in accessing the stdin, stdout, and stderr data streams, albeit in a slightly different scenario, in case you'd like to investigate this option further.

Regarding the option structure format names themselves, it's worth noting that the regular API standard pattern of four letters followed by four digits is not being observed. Why this is the case I don't know, but I did wonder why IBM has not enforced the common API standard, especially due to the ambiguity in the ZIP00100 format name—is the fourth byte an 'O' or a 'o'?

The fifth and final API parameter is, however, the good old standard API error structure, which has been discussed many times earlier, so I don't go into more detail on this topic here. I've included IBM's prototypes defining the Zip API interfaces below. Following installation of the PTFs referenced at the end of this article, you should find the RPG/IV prototypes as well as parameter structure definitions in the QZIPUTIL header file in QRPGLSRC in the system include library QSYSINC. The prototypes are included below:

D QzipZip	PR		EXTPROC
(*CWIDEN:'QzipZip')			
D filesToZip			LIKEDS
(Qlg_Path_Name_T) CONST			
D zipFileName			LIKEDS
(Qlg_Path_Name_T) CONST			
D formatName		8A	CONST
D zipOptions			LIKEDS
(Qzip_Zip_Options_T)			
D			CONST
D errorStruct		1000A	OPTIONS (*VARSIZE)
D QzipUnzip	PR		EXTPROC
(*CWIDEN:'QzipUnzip')			
D zipFileName			LIKEDS
(Qlg_Path_Name_T) CONST			
D unzipTargetPath...			
D			LIKEDS
(Qlg_Path_Name_T) CONST			
D formatName		8A	CONST
D unzipOptions			LIKEDS
(Qzip_Unzip_Options_T)			
D			CONST
D errorStruct		1000A	OPTIONS (*VARSIZE)

The Zip APIs are implemented by means of the QZIPUTIL service program located in library QSYS. The service program is written in ILE C++, hence IBM is following the convention of specifying either *CWIDEN or *CNOWIDEN in the prototype definition. In this case irrespective of no return value or parameters passed by value being present, which are normally considered the indicators for this practice.

Another issue to take into consideration is the fact that among the Zip APIs, error return messages are a number of messages supporting *CCHAR message data (a character string that can be converted). If data of this type is sent to a message queue that has a CCSID tag other than 65535 or 65534, the data is converted from the CCSID specified by the send function to the CCSID of the message queue.

To extend the *CCHAR convertible character support to the API error message handling in the two CPPs calling the Zip APIs, I employ the API error return message data structure format ERRCo200. For more information on this technique, please check out the article "APIs by Example: Using the ERRCo200 Data Structure," by following the link below.

Anyway, as for the two Zip File CL commands constructed on the basis of the corresponding Zip APIs, let's take a look at the Zip File (ZIPF) command prompt, in essence simply exposing the parameters supported by the QzipZip API:

```

                                Zip File (ZIPF)

Type choices, press Enter.

File to compress . . . . .

Compressed file name . . . . .

Verbose option . . . . .      *NONE          *NONE, *VERBOSE
Directory subtree . . . . .   *ALL           *ALL, *NONE
Comment . . . . .             *BLANK

```

You specify the file or directory to zip, as well as the zip file to create. Wildcard characters and pattern matching of the path name are not supported. The path can be an absolute path or a relative path name. All relative path names are relative to the current directory at the time when the ZIPF command is run. In addition to specifying whether a directory subtree should be included in the zip operation, you also have the option of associating a comment with the zip file being created as a result of the ZIPF command being run.

The complementary Unzip File (UNZIPF) command has the following prompt, which likewise exposes the parameters supported by the QzipUnzip API:

```

                                Unzip File (UNZIPF)

Type choices, press Enter.

Compressed file name . . . . .

Directory to place files . . . .

Verbose option . . . . .      *NONE          *NONE, *VERBOSE
Replace . . . . .             *NO            *YES, *NO

```

I've included the verbose option for both commands for completeness. Yet in order to actually employ this option, programming skills and efforts are involved, as mentioned earlier. For full documentation of the ZIPF and UNZIPF commands, please refer to both commands' online help text panel group. Note that the full path of the zipped object is placed in the specified directory when the object is decompressed and restored.

Also note that the CCSID of a zipped object is not preserved upon decompression, but rather reflects the job CCSID being in effect when the zip file is unzipped. This restriction needs to be considered in order to ensure that a decompressed text file's CCSID still reflects the file's actual content correctly. The Zip and Unzip APIs use the open-source zlib library to inflate and decompress the specified files, respectively. To learn more about the open-source zlib library, please follow the link at the end of this article pointing you to the zlib library home page.

This APIs by Example includes the following sources:

```
CBX240  -- RPGLE  -- Zip File - CPP
CBX240H -- PNLGRP -- Zip File - Help
CBX240V -- RPGLE  -- Zip File - VCP
CBX240X -- CMD    -- Zip File

CBX241  -- RPGLE  -- Unzip File - CPP
CBX241H -- PNLGRP -- Unzip File - Help
CBX241V -- RPGLE  -- Unzip File - VCP
CBX241X -- CMD    -- Unzip File

CBX240M -- CLP    -- Zip/Unzip File - Build Commands
```

To create all these command objects, compile and run the CBX240M CL program, following the instructions in the source header. You'll also find compilation instructions in the respective source headers.

PTFs Delivering 7.1 ZIP and UNZIP support:

[5770SS1-SI44777 - Zip and Unzip API on V7R1](#)

[5770SS1-SI44998 - Header files for QZIPUTIL service program](#)

Related articles and documentation:

[IBM i 7.1 Enhancements Optimize ISV Support Announcement](#)

[zlib Library Home Page](#)

[Barbara Morris, IBM: Converting from C prototypes to RPG prototypes](#)

[APIs by Example: Conversion of a Path Name](#)

[APIs by Example: Using the ERRCo200 Data Structure](#)

[Suppress PASE Output Messages \(stdin, stdout, stderr\)](#)

[Communicating Through a Pipe](#)

[Communicating Through a Pipe – Part 2](#)

[Don't Submit, Spawn!](#)

This article demonstrates the following UNIX-type APIs:

[Compress Files and Directories \(QzipZip\) API](#)

[Decompress an archive file \(QzipUnzip\) API](#)

[API Path name format](#)

[Error code parameter format](#)

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

Source URL: <http://iprodeveloper.com/rpg-programming/apis-example-zip-and-unzip-files-new-71-zip-api-support>