Susan Gantner
Jon Paris
Paul Tuohy
Gary Mullen-Schultz

RPG: APIs

Introduction

The number of APIs supplied with a system increases from release to release. After belonging to the realm of the “advanced” programmer, APIs are playing an ever-increasing part in our day-to-day applications, for example, you might want an application program to be able to retrieve invoices from a spooled file, convert those spooled files to PDFs, and e-mail them to the relevant customer. By supplying such functions in the form of APIs, i5/OS® makes the functions available to C, COBOL, Java™, and RPG programmers.

How many APIs are there? Well, back in 2003 there were over 1,500 APIs available to OS/400® developers. After that, we stopped counting!

Given the increased use of APIs, we decided to:

- Describe how to find APIs
- Explain how to interpret the API documentation
- Explain how to use APIs

What is an API?

The term *Application Programming Interface* (API) can be applied in many instances. By definition, an API is an application-supplied program or procedure that allows an application program, which is written in a high-level language, to access specific data or functions of the application without requiring access to the source code or requiring a detailed understanding of the functions' internal workings.

It is quite reasonable, and common, for any software application to provide APIs that provide access to “complex” portions of the application, for example, an application might provide an API that validates a product code or another API that calculates a price. Of course the supplier of the application provides adequate documentation on how to use the API.

IBM® supplies APIs that allow programmers to use specific data or functions of the operating system or a licensed program. The adequacy of the documentation is open to interpretation.

APIs provide functionality from the simple to the extremely complex. A lot of the APIs are not for the faint hearted, and if you are unfortunate enough to start out with the wrong API you will very possibly be wary of them forever.

Most programmers are introduced to APIs when they see them in an existing program. The most commonly used APIs are probably *Execute Command* (QCMDEXC), *Send Data Queue* (QSNDDTAQ) and *Receive Data Queue* (RCVDTAQ). You used APIs if you ever issued a call to any of these.

You must get to grips with the documentation and some of the methodologies that are used with APIs. But do not be discouraged. Although APIs may not be the easiest of items to get to grips with, they are far from being impossible to master.

Why use APIs?

APIs can allow access to system functions at a more detailed level than available with commands. APIs can also allow access to system information and functions that are not available through command line (CL) commands.

To make use of a lot of the new functionality in your applications, (for example, accessing data in directories in the IFS) you must make use of APIs.

Finding APIs

In this section, we provide information about where to go to find APIs and how to find APIs. The starting point for APIs is the Information Center, which you can access at:

`https://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp`

The API page

In the menu frame, expand Programming and APIs. The window, shown in Figure 1 on page 3, is displayed.

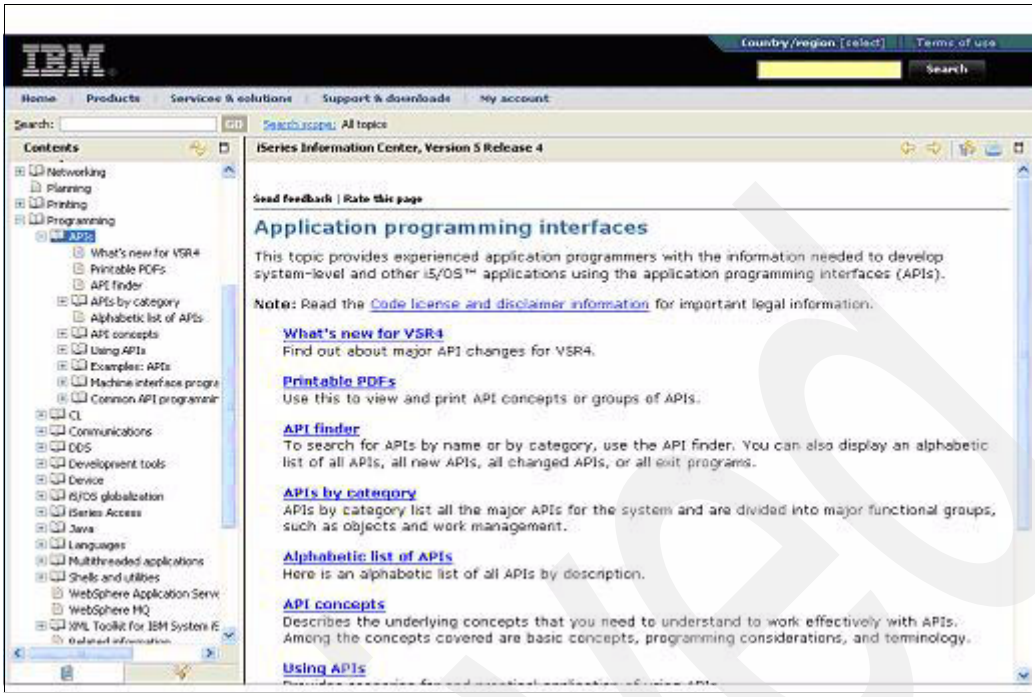


Figure 1 API documentation in the Information Center

The API page, Figure 1, provides three links for finding an API:

- API Finder
- APIs by Category
- Alphabetic List of APIs

If you have a preference for hard copy, you can take the link for Printable PDFs, download, and print the PDFs of your choice. At the time of writing, PDF documents are not available for Security or *UNIX®-Type* APIs.

There are also links for Concepts, Using APIs, and Examples, which we suggest that you review if you coded a couple of simpler APIs.

You are usually looking for an API for one of two reasons:

- You came across an API in a program, and you are wondering what it does.
- You want to perform a task, and you are wondering if there is an API that can help you achieve it.

Using the API Finder

The easiest way to find an API is to use the API Finder, shown in Figure 2 on page 4, which allows you to search for an API by:

- Its category (“Backup and Recovery,” “Message Handling,” “Object,” “Printing,” and so on)
- A search for its name or a search based on a partial description
- A group (by descriptive name being the most useful)

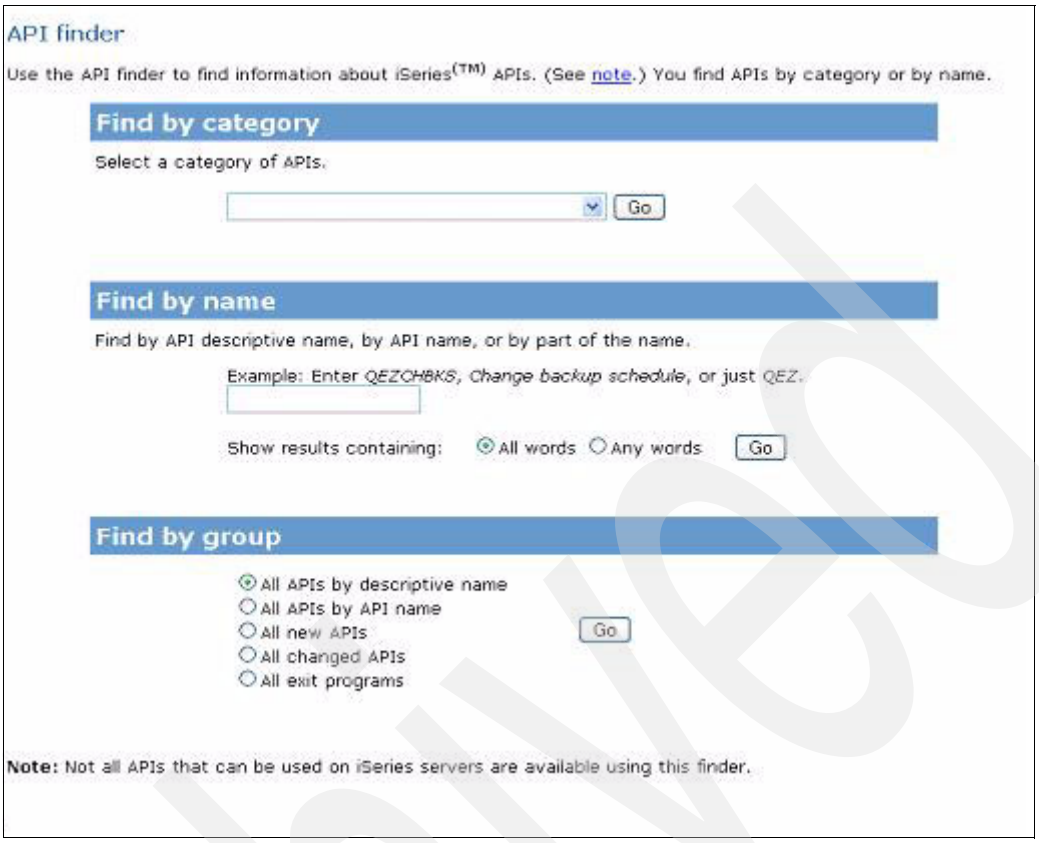


Figure 2 Using the API Finder function

Say you happen to come across a call to the QLIRLIBD API in a program, and you want to check its description:

1. In the Information Center (Figure 1 on page 3), click the **API Finder** link. The API Finder window is displayed (Figure 2).
2. In the Find by Name section, enter a value of QLIRLIBD, and click the **Go** button to reach a page, Figure 3, that lists all matches for your search. In our example, we received just one entry for Retrieve Library Description.

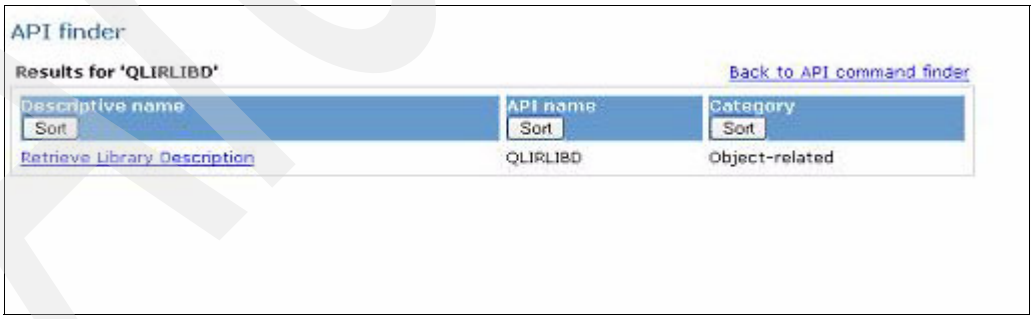


Figure 3 Result of searching for QLIRLIBD or Library Description

Perhaps you want to write a program that lists information about a library:

1. Go to the API Finder.
2. In the Find by Name section, enter a value of “Library Description”, and click the **Go** button to reach a page that lists all matches for your search. Again, there is just one entry for our example, Retrieve Library Description.

If you know that it is possible for a command to provide some of the information you are looking for, you can perform a general search for the command in the Information Center:

1. Go to the Information Center, and locate the Search field, which is at the top of the Web page(Figure 1 on page 3).
2. In the Search field, type the command name followed by the letters API (for example, DSPLIB API), and you may be lucky with the result set.

API Web sites

After you find the API you are looking for, try doing a general search on the Web for information about the API. There is a good chance that somebody else has already programmed this API, and, if you are lucky, they have a prototype and a working example available for you to download. An excellent example of this kind of web site is Think400 from Denmark, who at the time of writing, had nine pages of indexes to API examples. Think400 is located at:

`http://www.think400.dk/apier.htm`

Another excellent site is Thomas Bishop's AS400PRO.com, which provides not only examples but also provides a consolidated index of many other API examples. Access this information at:

`http://www.thomasbishop.com/servlet/sql.tipListInq?cat=API`

You should also check in QSYSINC/QRPGLESRC (requires System Openness Includes - option 13 of the base OS to be installed), for a source member with the same name as the API, which provides you with required data structure definitions; however, it does not provide prototypes, so you must code them yourself, or find them on the web. But, at least it provides you with a starting point. Unfortunately, the members in QSYSINC are mostly conversions from RPG III and use absolute notation for data structure definitions, use binary data types (as opposed to integers), use “short” field names, and do not take advantage of qualified data structures. RPGLE source members are not provided for all APIs, most notably the UNIX-Type APIs.

Tip: One of the best ways of performing a search for APIs on the IBM site is through Google. Simply specify “site:ibm.com” after your search phrase to specify that Google should only search ibm.com.

Types of APIs

There are three general types of available APIs:

- *Original Program Model* (OPM)
- *Integrated Language Environment*® (ILE)
- UNIX-Type

You can call all three types of APIs from an ILE program, but you can only call OPM APIs from an OPM program.

OPM APIs are provided as separate programs (dynamic calls), whereas ILE and UNIX-Type APIs are provided as procedures in service programs (static calls).

Some of the ILE APIs correspond to an OPM API, for example, you can call the Add Exit Program API to the program QUSADDEP or call to the procedure **`QusAddExitProgram`**. Both have identical parameters and are described on the same page of the API reference.

There are a number of differences between the three types of APIs, most notably in the way they are described in the Information Center. The descriptions of the OPM and ILE APIs are written in a generic non-language specific format, but the UNIX-Type APIs are described from the perspective of the C programmer.

Deciphering the documentation

Probably the best way to start with understanding the documentation is to look at a couple of examples: the Execute Command (QCMDEXC) and the Retrieve Library Description (QLIRLIBD) APIs. But first you must be familiar with the description of parameter data types for APIs.

Parameter data types

You must remember that the descriptions of the parameters are supposed to be language neutral, so do not fall into the trap of doing a direct translation of the data type and length of a parameter.

Some of the descriptions are straight forward and can be translated directly, like CHAR(10) (10A in RPG) or PACKED(15,5) (15P 5 in RPG). But others are not straightforward.

A definition of CHAR(*) indicates that the length of the parameter string varies. There is usually a corresponding parameter to indicate the length of the parameter string used. So, you must determine the length of the string and specify that length in the corresponding second parameter (you see this with QCMDEXC).

Yet, there is the definition that causes the most confusion - BINARY(4), which does not translate to 4B 0 in RPG. BINARY(4) indicates a 4 byte binary integer, whereas 4B 0 in RPG indicates a 2 byte binary field that contains four digits.

Actually, you should avoid the B data type in RPG altogether, even though most of the examples in the Information Center still use them. This is a throwback to the days of RPG III when you could only define binary fields and not integers. A 4 byte binary integer can store a value in the range -2,147,483,648 to 2,147,483,647, whereas a 4 byte binary field in RPG (defined as 9B 0) can only hold a value in the range -999,999,999 to 999,999,999. Integers are also more efficient than binary data types. Refer to the FAQ on www.midrange.com for more information about the meaning of BINARY(4) in API documentation.

http://faq.midrange.com/data/cache/24.html

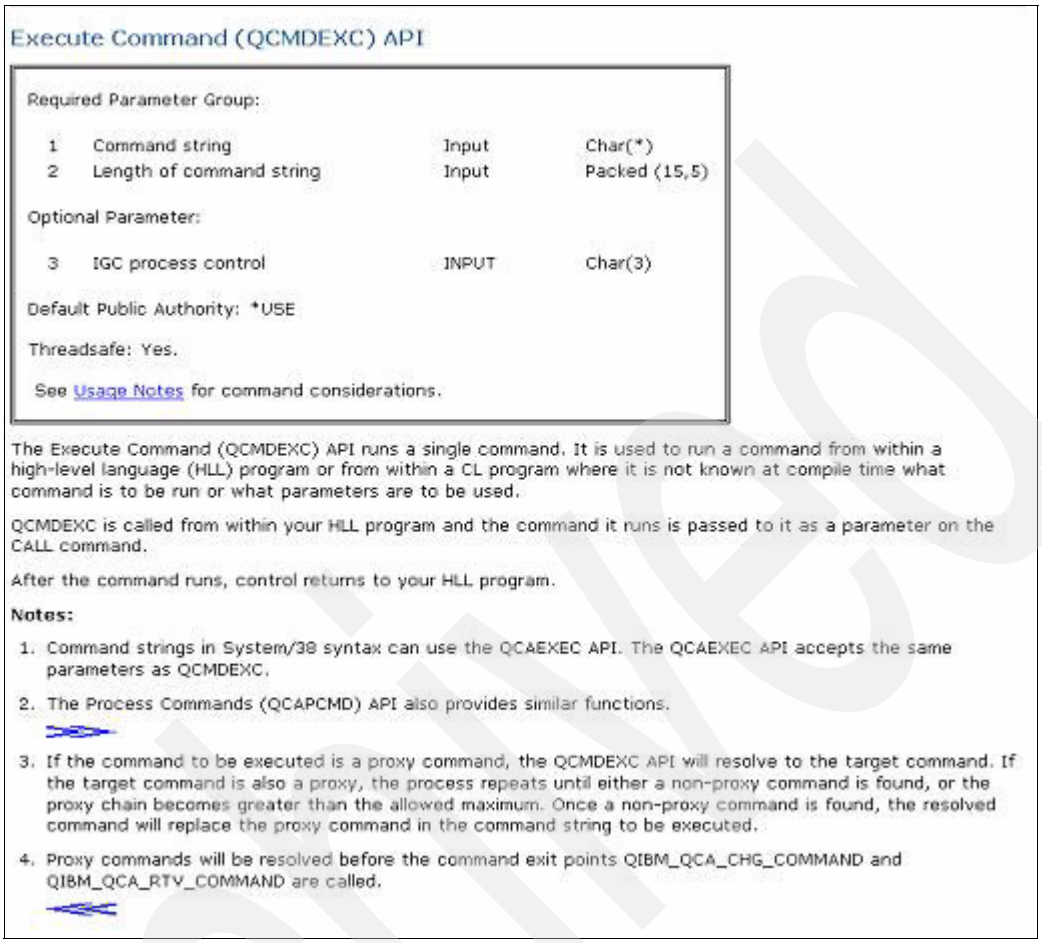
So a definition of BINARY(4) translates to 10I 0 in RPG, and a definition of BINARY(4) UNSIGNED translates to 10U 0.

Execute command (QCMDEXC)

The QCMDEXC API is the API you have seen most used in application programs.

The documentation for QCMDEXC starts with a general description of the parameters and a description of what the API does (Figure 4 on page 7).

Figure 4 is an example of what the QCMDEXC API does.



Execute Command (QCMDEXC) API

Required Parameter Group:

1	Command string	Input	Char(*)
2	Length of command string	Input	Packed (15,5)

Optional Parameter:

3	IGC process control	INPUT	Char(3)

Default Public Authority: *USE

Threadsafe: Yes.

See Usage Notes for command considerations.

The Execute Command (QCMDEXC) API runs a single command. It is used to run a command from within a high-level language (HLL) program or from within a CL program where it is not known at compile time what command is to be run or what parameters are to be used.

QCMDEXC is called from within your HLL program and the command it runs is passed to it as a parameter on the CALL command.

After the command runs, control returns to your HLL program.

Notes:

1. Command strings in System/38 syntax can use the QCAEXEC API. The QCAEXEC API accepts the same parameters as QCMDEXC.
2. The Process Commands (QCAPCMD) API also provides similar functions.
3. If the command to be executed is a proxy command, the QCMDEXC API will resolve to the target command. If the target command is also a proxy, the process repeats until either a non-proxy command is found, or the proxy chain becomes greater than the allowed maximum. Once a non-proxy command is found, the resolved command will replace the proxy command in the command string to be executed.
4. Proxy commands will be resolved before the command exit points QIBM_QCA_CHG_COMMAND and QIBM_QCA_RTV_COMMAND are called.

Figure 4 Parameters and functional description for the QCMDEXC API

The parameter descriptions of the QCMDEXC API consist of:

- Required Parameter Group, which are the parameters that MUST be passed on the call.
- An Optional Parameter Group, which are parameters that may or may not be passed on the call. You must specify all parameters in an Optional Parameter Group even if you only need to use one. Some APIs can have more then one optional parameter group (for example QRCVDTAQ has two and QSNDDTAQ has three).
- A short description of each parameter, which we describe in detail later in the paper. Always make sure that you read the detailed description of the parameter, especially with CHAR, since it could actually be a structure that is being defined as opposed to a simple string.
- An indication of how the parameter is used by the API.
  - INPUT means that the parameter is input to the API, but the API does not change it.
  - OUTPUT means that the API returns a value in the parameter but does not care about the value input.
  - I/O means that the value of the parameter is INPUT to the API and that a value is OUTPUT from the API.
- A description of the type and length of the parameter. There then follows a description of what the API does, how it works and special notes.

The documentation continues with a description of any required authorities required to use the API and any locks that may be caused when using it, as shown in Figure 5.

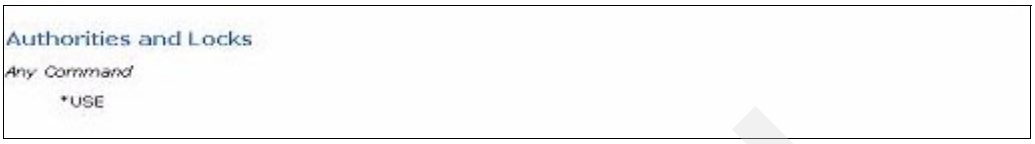


Figure 5 Authorities and Locks for the QCMDEXC API

The Required Parameter Group, shown in Figure 6, gives a detailed description of each of the required parameters. The first parameter is defined as CHAR(*), and the value of the second parameter indicates the actual length of the first parameter. This is quite a common feature that many APIs use.

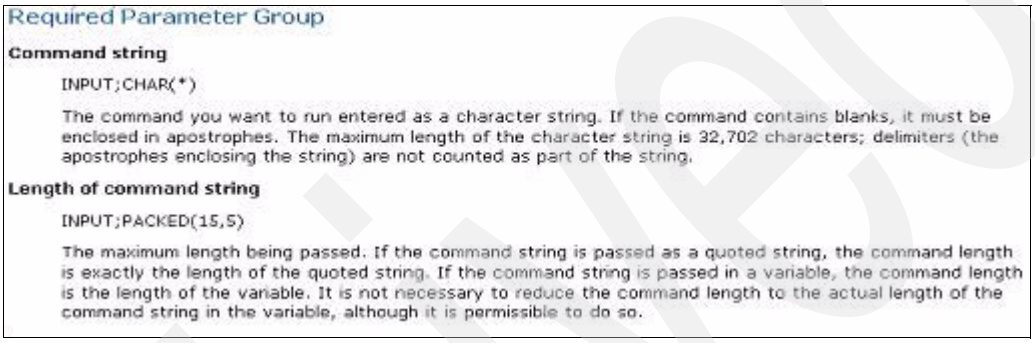


Figure 6 Required Parameter Group for the QCMDEXC API

The Optional Parameter Group, shown in Figure 7, gives a detailed description of each of the optional parameters. The optional parameter for QCMDEXC, if specified, must have a value of 'IGC' to indicate that DBCS is to be accepted.

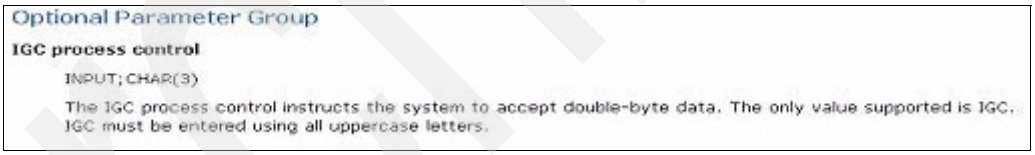
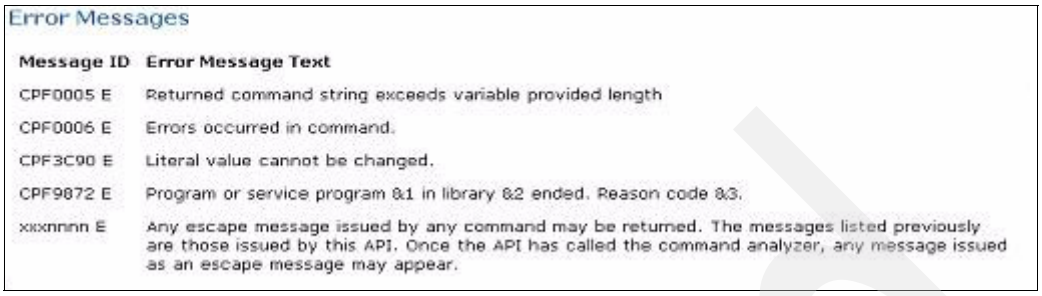


Figure 7 Optional Parameter Group for the QCMDEXC API

The Usage Notes, shown in Figure 8, provide any warnings about when or where you must not use the API.

Figure 8 Usage Notes for the QCMDEXC API

Error Messages, shown in Figure 9, list the messages that you can issue as a result of calling the API. These messages are in the program status data structure.



Error Messages

Message ID	Error Message Text
CPF0005 E	Returned command string exceeds variable provided length
CPF0006 E	Errors occurred in command.
CPF3C90 E	Literal value cannot be changed.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
xxxnnnn E	Any escape message issued by any command may be returned. The messages listed previously are those issued by this API. Once the API has called the command analyzer, any message issued as an escape message may appear.

Figure 9 Error Messages for the QCMDEXC API

Lastly, there is the line at the end of the page, shown in Figure 10, that indicates when the API was introduced, prior to V1R3 for QCMDEXC.

Figure 10 Release information for the QCMDEXC API

Next, we look at an example of using the QCMDEXC API.

Using the QCMDEXC API

Example 1 shows that the member, STDAPIINFO, contains the prototype for the QCMDEXC API.

Example 1 STDAPIINFO contains the prototype for QCMDEXC API

```
(1)  D ExecuteCommand  PR                  ExtPgm('QCMDEXC')
(2)  D  Command                  3000     Const Options(*VarSize)
     D  CommandLen                 15   5 Const
(3)  D  IGCProcess                  3     Const Options(*NoPass)
```

The following list contains the main points in Example 1. The numbers in this list correspond with the numbering in Example 1:

1. Each of the parameters is defined with a CONST keyword, since they are all INPUT parameters and you are not concerned about a value being returned. This allows us the freedom to pass constants, expressions, or fields of slightly different definitions as parameters to the API. Refer to section 3.6.1 of the IBM Redbooks® publication *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402 for more information about prototype keywords.
2. The command parameter is defined with OPTIONS(*VARSIZE) to indicate that any length up to 3000 is acceptable (the length of 3,000 is arbitrary, it must be enough for the longest command you might issue).
3. The IGC process parameter is defined with OPTIONS(*NOPASS) to indicate that it is an optional parameter.

The program API01 is an example of a program that requests a library name and calls the Display Library (DSPLIB) command to list its contents, as shown in Example 2.

Example 2 DSPLIB API example

```
H Option(*SrcStmt:*NoDebugIO)

      /Copy APISRC,STDAPIINFO

     D MyCommand       S             50
     D Library         S             10

     D ProgramInfDS   SDS                   NoOpt
     D                                      Qualified
(1)  D  MsgId                          7    Overlay(ProgramInfDS:40)
      /Free
(2)        Dsply 'Library Name: ' ' ' Library;
(3)        MyCommand = 'DSPLIB ' + %Trim(Library) + ' *PRINT';
           Monitor;
(4)           ExecuteCommand(MyCommand:%Len(MyCommand));
           On-Error;
(5)           Dsply ProgramInfDS.MsgId;
           EndMon;
           *InLR = *On;
      /End-Free
```

Create this program by using the command:

```
CRTBNDRPG PGM(REDBOOK/API01) SRCFILE(REDBOOK/APISRC)
```

A successful call to API01 results in a spooled file that lists the contents for the library name that you entered. The following list contains the main points of Example 2. The numbers in this list correspond with the numbering in Example 2:

1. The message ID in the program status data structure identifies the message ID for any failed command. If the message ID alone is insufficient for your error handling needs, you can use the alternative QCAPCMD, which uses the standard API error structure that we describe in the following section.
2. A library name is entered.
3. The required command string is constructed.
4. The QCMDEXC API is called to issue the command. The use of “%LEN BIF” for the second parameter on the call is allowed because of the CONST keyword.
5. If the call to QCMDEXC ends in error, the message ID from the program status data structure is displayed. In Example 2, we use a Monitor group as opposed to an E extender on a CallP.

Error structure

The error structure is common to a lot of APIs and is a means by which an API can indicate that an error occurred and provide information about the error. Example 3 is the definition of an error structure defined in the member STDAPIINFO and used in the examples in this chapter.

Example 3 Error structure definition in STDAPINFO

```
     D APIError        DS                  Qualified
(1)  D  BytesProvided                10I 0 inz(%size(APIError))
(2)  D  BytesAvail                   10I 0 inz(0)
(3)  D  MsgId                         7A
(4)  D                                1A
(5)  D  MsgData                     240A
```

The following definitions, which are numbered to correspond with the numbering in Example 3, are:

1. **`BytesProvided`** tells the API the size of the error structure, which determines how many bytes of information, regarding an error, are returned by the API. The %size BIF is used to initialize the value of the field to the size of the error structure.
2. **BytesAvail** is set by the API and indicates how many bytes, regarding an error, were returned by the API. The value of this field is greater than 0 if the API detected an error, so check the value of this field if you need to know if there was an error.
3. **MsgID** is the seven character message ID that identifies the error. It corresponds to one of the Error Messages listed in the documentation for the API.
4. The unnamed “filler” field is required and your program should not reference it.
5. **MsgData** is the variable message data for the message. This is not the complete message text, but it is the variable portion of the message text, for example, the equivalent of the information you provide for the Message Data (MSGDTA) parameter on the Send Program message (SNDPGMMSG) command. The size of 240 is arbitrary but is large enough to accommodate most situations without wasting storage.

The error code parameter is optional for some APIs, particularly the older ones. If you do not include the error code parameter, the API returns both diagnostic and escape messages to the caller. If the parameter is coded, diagnostic messages are not returned. Exception messages are returned to the caller based on the length available.

If you specify a **`BytesProvided`** length as zero, when an error occurs, an exception is returned to the caller.

If you specify a **`BytesProvided`** length of eight, (actually any length from 8 - 11 has the same effect) the ID of the exception message is placed in the MsgId field. No exception is signaled to the application.

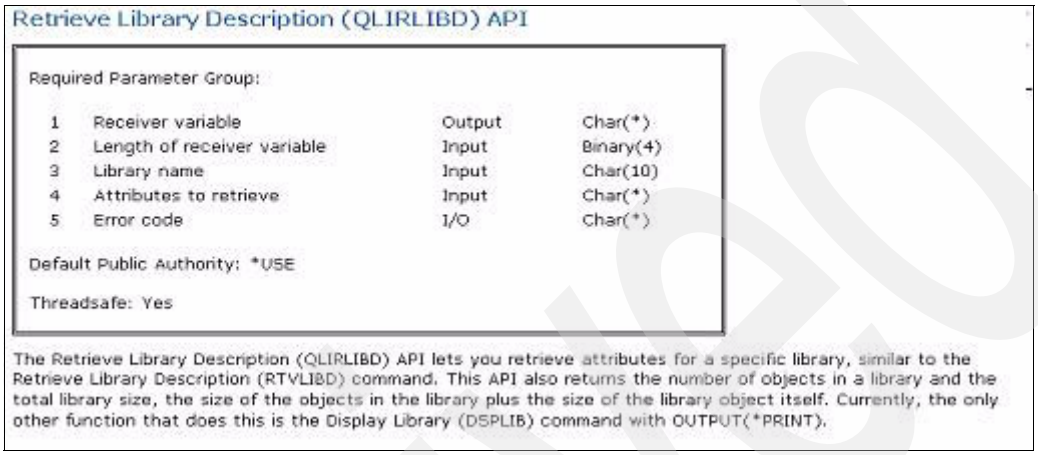
If you specify a **`BytesProvided`** length of 12 or more, then if an error occurs, in addition to the MsgId, the MsgData is also filled in.

Even if you ultimately plan to have the API return the full set of exception data and act appropriately, you might find it useful during the early stages of testing to set the length to zero. Doing so causes your job to immediately halt if there is an error and allows you to look at the job log and review the messages. Later, as you add the logic to handle exceptions, you can set the length parameter to an appropriate value.

Retrieve Library Description (QLIRLIBD)

Now that you know the basic layout of the documentation, we can examine an API that has a slightly more complex parameter structure. The QLIRLIBD API returns up to eight items of information about a library, for example, text description, size of library, and the number of objects in library.

Figure 11 shows the description and parameter for the QLIRLIBD API.


Retrieve Library Description (QLIRLIBD) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	Library name	Input	Char(10)
4	Attributes to retrieve	Input	Char(*)
5	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: Yes

The Retrieve Library Description (QLIRLIBD) API lets you retrieve attributes for a specific library, similar to the Retrieve Library Description (RTVLIBD) command. This API also returns the number of objects in a library and the total library size, the size of the objects in the library plus the size of the library object itself. Currently, the only other function that does this is the Display Library (DSPLIB) command with OUTPUT(*PRINT).

Figure 11 Parameters for the QLIRLIBD API

The member STDAPIINFO contains the corresponding prototype for the QLIRLIBD API.

Example 4 QLIRLIBD API

```
D RetrieveLibraryInfo...
     D                 PR                  ExtPgm('QLIRLIBD')
(1)  D  Receiver                65535    Options(*VarSize)
(2)  D  ReceiverLen                10I 0 Const
(3)  D  Library                    10    Const
(4)  D  RtvAttributes              50    Const Options(*VarSize)
     D  Error                            LikeDS(APIError)
```

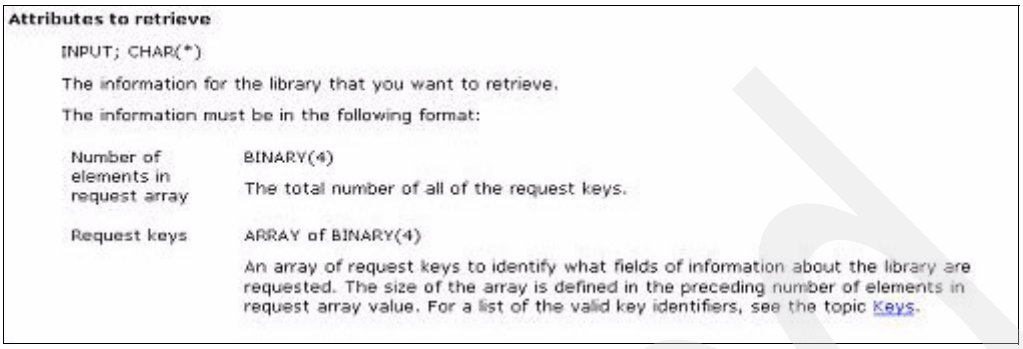
The parameters, which are numbered to correspond with the numbers in Example 4, are:

1. The information returned by QLIRLIBD (more in a moment).
2. The length of the first parameter.
3. The name of the library.
4. The attributes to retrieve.

The **Receiver** and **RtvAttributes** parameters deserve further examination.

Retrieve Attributes

The documentation for QLIRLIBD, shown in Figure 12, describes the Attributes to Retrieve parameter as a structure.



Attributes to retrieve

INPUT; CHAR(*)

The information for the library that you want to retrieve.

The information must be in the following format:

Number of elements in request array	BINARY(4) The total number of all of the request keys.
Request keys	ARRAY of BINARY(4) An array of request keys to identify what fields of information about the library are requested. The size of the array is defined in the preceding number of elements in request array value. For a list of the valid key identifiers, see the topic Keys.

Figure 12 Description of the Attributes to Retrieve parameter

The member STDAPIINFO contains the corresponding definition of the structure used to identify the required attributes.

Example 5 Definition of the structure that identifies the required attributes

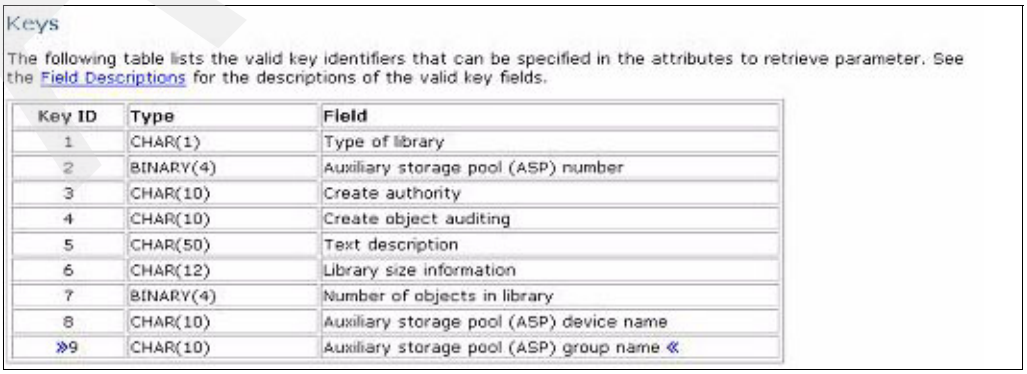
```
D GetLibraryAttributes...
(1)  D                  DS                  Qualified Based(DummyPtr)
(2)  D  NumberOfkeys                10I 0

(3)  D  RequestKeys                 10I 0 Dim(15)
```

The following important points are numbered to correspond with the numbering in Example 5:

1. Since this data structure (and all other structures in STDAPIINFO) is for definition purposes only, the data structure is based on a “dummy pointer” that is never set. This means that the data structure does not occupy any memory in a program in which it is included. When using the QLIRLIBD API, programs must use the LIKEDS keyword to define a data structure with the same definition.
2. **`NumberOfKeys`** is the number of elements in **`theRequestKeys`** array that contain a requested attribute key.
3. Each element of the **`RequestKeys`** array has a key ID (a number one to nine) that identifies the attribute required.

Figure 13 shows the documentation for the available keys. Please note that key ID 9 is only available from V5R4 onward.



Keys

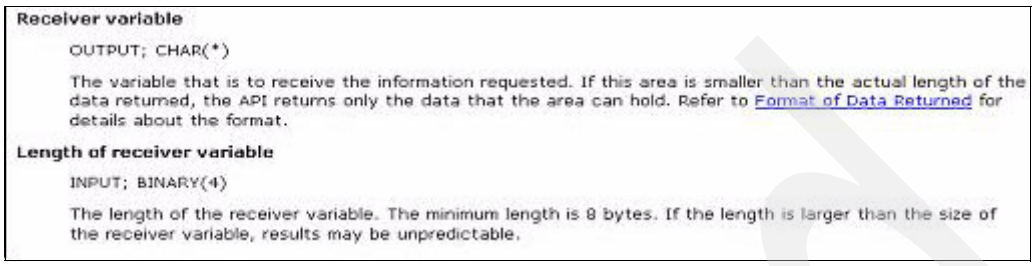
The following table lists the valid key identifiers that can be specified in the attributes to retrieve parameter. See the Field Descriptions for the descriptions of the valid key fields.

Key ID	Type	Field
1	CHAR(1)	Type of library
2	BINARY(4)	Auxiliary storage pool (ASP) number
3	CHAR(10)	Create authority
4	CHAR(10)	Create object auditing
5	CHAR(50)	Text description
6	CHAR(12)	Library size information
7	BINARY(4)	Number of objects in library
8	CHAR(10)	Auxiliary storage pool (ASP) device name
»9	CHAR(10)	Auxiliary storage pool (ASP) group name «

Figure 13 Description of the Attribute Keys

Receiver parameter

The documentation for QLIRLIBD, shown in Figure 14, specifies that the receiver is actually a format that is further described at “Format of Data Returned.” The second parameter must identify the length of the structure you are providing as the first parameter.



Receiver variable

OUTPUT; CHAR(*)

The variable that is to receive the information requested. If this area is smaller than the actual length of the data returned, the API returns only the data that the area can hold. Refer to Format of Data Returned for details about the format.

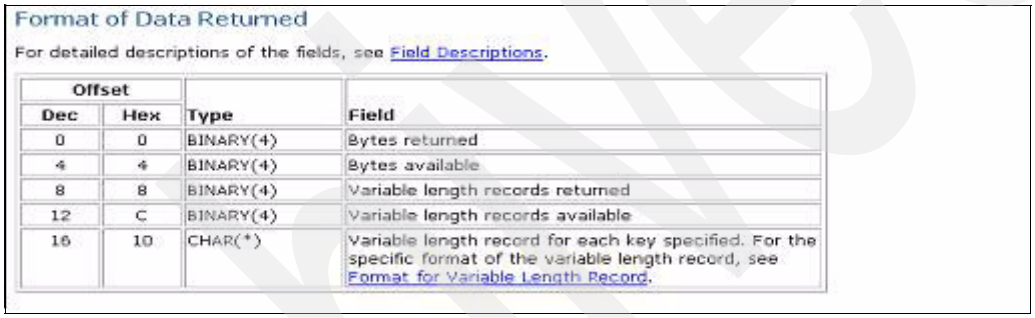
Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. The minimum length is 8 bytes. If the length is larger than the size of the receiver variable, results may be unpredictable.

Figure 14 Description of the Receiver and Receiver Length parameters

The documentation describes the format of data returned, shown in Figure 15, as a structure which, in turn, contains yet more variable length data that is further described at “Format for Variable Length Records.”



Format of Data Returned

For detailed descriptions of the fields, see Field Descriptions.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Variable length records returned
12	C	BINARY(4)	Variable length records available
16	10	CHAR(*)	Variable length record for each key specified. For the specific format of the variable length record, see Format for Variable Length Record.

Figure 15 Description of the Format of Data Returned

The receiver variable contains information in the format shown in Figure 16. The number of variable length records and the size of each record is dependent on the number of attributes and attributes requested.

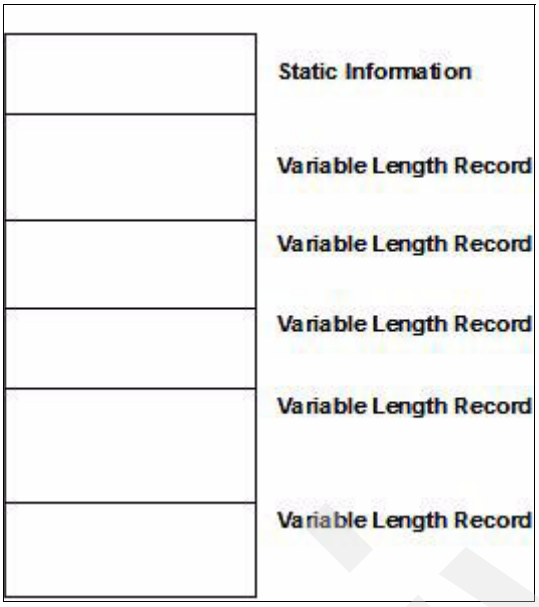


Figure 16 Format of the Receiver Variable Returned

The member STDAPIINFO contains the corresponding definition of the structure returned by the API.

Example 6 Definition of the structure that the API returns

```
D LibraryInfoReceiver...
     D                 DS                  Qualified Based(DummyPtr)
(1)  D  BytesReturned                 10I 0
(2)  D  BytesAvail                    10I 0
(3)  D  VarLenRecReturned...
     D                                10I 0
(4)  D  VarLenRecAvailable...
     D                                10I 0
(5)  D  VarData                     512
```

The subfields returned in the data structure, which are numbered to correspond with the numbering in Example 6, are:

1. **BytesReturned** indicates the number of actual bytes returned by the API.
2. **BytesAvail** indicates the number of bytes that the API could have returned. The value of **BytesAvail** can be greater than **BytesReturned** if the amount of data that the API can return exceeded the size of the format that was specified for the receiver parameter.
3. **VarLenRecReturned** indicates the number of information records that the API returned.
4. **VarLenRecAvail** indicates the number of information records available. As with **BytesReturned** and **BytesAvail**, any difference between the two is dependent on the size of the format that is specified for the receiver parameter.
5. **VarData** is the data returned, and it contains one or more variable length records, which are described in the documentations, as shown in Figure 17 on page 16.

Format for Variable Length Record

For detailed descriptions of the fields, see Field Descriptions.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Length of returned data
4	4	BINARY(4)	Key identifier
8	8	BINARY(4)	Size of field
12	C	CHAR(*)	Field value
		CHAR(*)	Reserved

Figure 17 Description of the Format of Variable Length Record

The member STDAPIINFO contains the corresponding definition of the variable length record.

Example 7 Definition of the variable length record

```
D FormatVariableRecord...
     D                 DS                  Qualified Based(DummyPtr)
(1)  D  LenRetData                   10I 0
(2)  D  KeyId                        10I 0
(3)  D  FldSize                      10I 0
(4)  D  FldValue                     50
     D   FldValue12                  12    OverLay(FldValue)
     D   FldValue10                  10    OverLay(FldValue)
     D   FldValue1                    1    OverLay(FldValue)
     D   FldValueInt                 10I 0 OverLay(FldValue)
```

This format only contains the actual data returned, for example, each instance cannot occupy 62 bytes of storage. The main points to note, which are numbered to correspond with the numbering in Example 7, are:

1. **LenRetData** is the actual length of the variable length format.
2. **KeyId** is the key for which a value is being returned.
3. **FldSize** (Size of Field) is the size of the returned field.
4. **FldValue** (Field Value) is the value returned for the **KeyId**; however, the size of the field value is different depending on the value of the **KeyId** (for example, the length of 1 returned for a **KeyId** with a value of 1—length of 10 returned for a **KeyId** with a value of 3); therefore, **FldValue** is overlaid by fields that correspond to the different returned lengths. Refer to Figure 13 on page 13 for the type of field that is returned for each attribute.

The documentation specifies that the field value that is returned for the library size (type 6) is another structure in the format shown in Figure 18.

Library Size Information Format

The following table shows the layout of the library size information key. For detailed descriptions of the fields, see Field Descriptions.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Library size
4	4	BINARY(4)	Library size multiplier
8	8	CHAR(1)	Information status
9	9	CHAR(3)	Reserved

Figure 18 Description of the Library Size Information Format

Example 8 shows that the member STDAPIINFO contains the corresponding definition of the Library Size Information Format.

Example 8 Definition of the Library Size Information Format

```
D LibrarySizeInformation...
     D                 DS                  Qualified Based(DummyPtr)
(1)  D  LibrarySize                  10I 0
(2)  D  LibraryMult                  10I 0
(3)  D  InfoStatus                    1
     D                                3
```

The data structures, which are numbered to correspond with the numbering in Example 8, consists of:

1. **LibrarySize** is the base size of the library.
2. **LibraryMult** is used to multiply the size to get the actual size. The multiplier is 1 for any size less then 1,000,000,000.
3. **InfoStatus** contains a value of “1” to indicate that all objects in the library were used in determining the size.

From the information in this section, you should have some idea of the “complexity” of how an API returns information—a format that contains one or more variable length formats at least one of which may contain another format.

In the next section, we look at an example of using QLIRLIBD.

Using the QLIRLIBD API

The program API02 uses the QLIRLIBD API to retrieve and list requested attributes for the requested library, as shown in Example 9.

Example 9 QLIRLIBD API

```
     H Option(*SrcStmt:*NoDebugIO)

(1)   /Copy APISRC,STDAPIINFO

(2)  D GetAttributes   DS                  LikeDS(GetLibraryAttributes)
(3)  D                                     Inz

(2)  D FormatReturn    DS                  LikeDS(LibraryInfoReceiver)

     D pFormatVarRec   S                   *
(2)  D FormatVarRec    DS                  LikeDS(FormatVariableRecord)
(4)  D                                     Based(pFormatVarRec)

(2)  D LibSizInf       DS                  LikeDS(LibrarySizeInformation)

     D i               S                   5I 0
     D DspData         S                  52
     D Library         S                  10
     D KeysIn          S                   9
     D ForKeys         S                   9    Varying
     D LOWER           C                        'abcdefghijklmnopqrstuvwxyz'
```

```
     D UPPER           C                   'ABCDEFGHIJKLNMOPQRSTUVWXYZ'

      /Free
(5)       Dsply 'Library: ' ' ' Library;
          DspLy 'Keys (1 to 9): ' ' ' KeysIn;
          Library = %XLate(LOWER: UPPER: Library);
          ForKeys = %Trim(KeysIn);

(6)       For i = 1 to %Len(ForKeys);
             GetAttributes.RequestKeys(i) =
                       %Int(%SubSt(ForKeys:i:1));
          EndFor;

(7)       GetAttributes.NumberOfkeys = %Len(ForKeys);

(8)       RetrieveLibraryInfo( ForMatReturn
                             : %Len(FormatReturn)
                             : Library
                             : GetAttributes
                             : APIError);

(9)       If APIError.BytesAvail > 0;
              Dsply APIError.MsgId;
              Dsply %Subst(APIError.MsgData:1:52);
              *InLR = *On;
              Return;
          EndIf;

(10)      pFormatVarRec = %Addr(FormatReturn.VarData);
(11)      For i = 1 to FormatReturn.VarLenRecReturned;
(12)          Select;
                  When FormatVarRec.KeyId = 1;
                      DspData = '1 Type: '
                                + FormatVarRec.FldValue1;
                  When FormatVarRec.KeyId = 2;
                      DspData = '2 ASP No: '
                                + %Char(FormatVarRec.FldValueInt);
                  When FormatVarRec.KeyId = 3;
                      DspData = '3 Create Authority: '
                                + FormatVarRec.FldValue10;
                  When FormatVarRec.KeyId = 4;
                      DspData = '4 Create Object Auditing: ' +
                                  FormatVarRec.FldValue10;
                  When FormatVarRec.KeyId = 5;
                      DspData = '5 '
                                  + FormatVarRec.FldValue;
                  When FormatVarRec.KeyId = 6;
                      LibSizInf = FormatVarRec.FldValue12;
                      DspData = '6 Size: '
                                  + %Char(LibSizInf.LibrarySize)
                                  + ' ' + %Char(LibSizInf.LibraryMult)
                                  + ' ' + LibSizInf.InfoStatus;
                  When FormatVarRec.KeyId = 7;
                      DspData = '7 Objects: '
                                 + %Char(FormatVarRec.FldValueInt);
```

```
                When FormatVarRec.KeyId = 8;
                    DspData = '8 ASP Device: '
                              + FormatVarRec.FldValue10;
                When FormatVarRec.KeyId = 9;
                    DspData = '9 ASP Group: '
                              + FormatVarRec.FldValue10;
            EndSl;

            DspLy DspData;
(13)        pFormatVarRec = pFormatVarRec + FormatVarRec.LenRetData;
        EndFor;

        *InLR = *On;
    /End-Free
```

Create this program by using the command:

```
CRTBNDRPG PGM(REDBOOK/API02) SRCFILE(REDBOOK/APISRC)
```

The main points of interest, which are numbered to correspond with the numbering in Example 9 on page 17, are:

1. Copy directive includes the required prototype and format definitions.
2. LIKEDS defines data structures for the Get Attributes, Receiver, Variable Length Record, and Library Size Information formats.
3. The **GetAttributes** data structure is initialized to ensure that the integers in the data structure are initialized properly.
4. The data structure for the format of the variable length record is based on a pointer. The data structure is overlaid onto each returned attribute record.
5. The library name and key attributes are input. The library name is converted to uppercase (the library name is case sensitive). The string of attribute keys is trimmed and moved to a variable length field (just to make the following loop through the attributes that little bit easier).
6. The requested attributes are placed in the array of request keys.
7. The number of request keys is set.
8. The QLIRLIBD API is called.
9. The value of the bytes available in the error code is checked. If it is greater then zero (for example, there is an error), the error message returned is displayed and the program ends. To see this working, specify an invalid library name and duplicate attributes or an invalid attribute key.
10. The program sets the pointer for **FormatVarRec** equal to the address of **VarData** in **FormatVarRec**, for example, **FormatVarRec** now overlays the start of the variable data returned by the QLIRLIBD API, so it is overlaying the first variable length record returned.

11. **VarLenRecRet** (the number of records returned) is the basis for a loop.
12. The program displays the relevant data returned based on the value of the Key ID. Look again at the definition of **FormatVarRec** and note how **FldValue** is overlaid by fields of different lengths and types. It is important that you use the right lengths for the different key IDs because they are the actual lengths returned—remember, it is a variable length record. Also note how the value for Key ID 6 (Library Size) is moved to the **LibSizInf** data structure for further breakdown.
13. The interesting bit is at the end of the FOR loop. You just finished with the record for one key, and you now want **FormatVarRec** to overlay the next variable length record returned. Simply add the length of the current variable length record to the current value of the pointer.

Table 1 shows the contents of the receiver variable that contains the variable length records for attributes one to eight.

Table 1 Sample contents of the Receiver variable for QLIRLIBD

Field name	Start position	Description	Value
BytesReturned	1	Bytes Returned	222
BytesAvail	5	Bytes Available	222
VarLenRecReturned	9	Variable Length Records Returned	8
VarLenRecAvailable	13	Variable Length Records Available	8
LenRetData	17	Length of Returned Data	16
KeyId		Key Id	1
FldSize		Field Size	1
FldValue		Field Value	'0'
LenRetData	33	Length of Returned Data	16
KeyId		Key Id	2
FldSize		Field Size	4
FldValue		Field Value	1
LenRetData	49	Length of Returned Data	24
KeyId		Key Id	3
FldSize		Field Size	10
FldValue		Field Value	'*SYSVAL '

Field name	Start position	Description	Value
LenRetData	73	Length of Returned Data	24
KeyId		Key Id	4
FldSize		Field Size	10
FldValue		Field Value	'*SYSVAL '
LenRetData	97	Length of Returned Data	64
KeyId		Key Id	5
FldSize		Field Size	50
FldValue		Field Value	'Sample Development Library'
LenRetData	161	Length of Returned Data	24
KeyId		Key Id	6
FldSize		Field Size	12
FldValue		Field Value	892928 1 '1'
LenRetData	185	Length of Returned Data	16
KeyId		Key Id	7
FldSize		Field Size	4
FldValue		Field Value	6
LenRetData	201	Length of Returned Data	22
KeyId		Key Id	8
FldSize		Field Size	10
FldValue		Field Value	'*SYSBAS '

Example 10 shows the corresponding information that the program displays.

Example 10 Information that the program displays

```
DSPLY  Library:
redbook
DSPLY  Keys (1 to 9):
12345678
DSPLY  1 Type: 0
DSPLY  2 ASP No: 1
```

```
DSPLY  3 Create Autority: *SYSVAL
DSPLY  4 Create Object Auditing: *SYSVAL
DSPLY  5 Sample Development Library
DSPLY  6 Size: 892928 1 1
DSPLY  7 Objects: 6
DSPLY  8 ASP Device: *SYSBAS
```

The List APIs

Many of the APIs return lists of information, which means that the API returns a variable amount of information, for example, the amount of information that the List Objects (QUSLOBJ) API returns is dependant on the number of objects being listed.

The *List APIs* are not a category of API (you will not find them on the "APIs by Category" Web page); instead, they refer to the way that the API works. They all work in the exact same way.

To create a user space:

1. Call the List API that provides the name of the user space. The required information is placed in the user space.
2. Process the information in the user space.
3. Repeat steps 2 and 3 as often as required.
4. Delete the user space when you are finished. Alternatively, create the user space in QTEMP so that it automatically disappears when the job ends.

You need to make use of user spaces when you are using the List APIs.

User spaces

Section 5.3 of the IBM Redbooks publication *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402 gives a fairly detailed description of user spaces, so we are not going to repeat the process here. But it is a good idea to write a couple of subprocedures to handle the user spaces required by the List APIs.

The member STDAPIINFO contains prototypes for the **CreateListSpace** and **DeleteListSpace** subprocedures, as shown in Example 11.

Example 11 CreateListSpace and DeleteListSpace subprocedures of STDAPIINFO

```
D CreateListSpace...
     D                 PR              *   ExtProc('CREATELISTSPACE')
     D  SpaceName                    10A   Const

     D DeleteListSpace...
     D                 PR                  ExtProc('DELETELISTSPACE')
     D  SpaceName                    10A   Const
```

Both subprocedures accept the name of a user space and the CreateListSpace subprocedure returns a pointer to the address of the user space in memory.

The member STDAPIINFO also contains prototypes for the User List APIs to Create a User Space (QUSCRTUS), Change User Space Attributes (QUSCUSAT), Retrieve Pointer to User

Space (QUSPTRUS) and Delete a User Space (QUSDLTUS) along with the definition of the attribute structure that QUSCUSAT requires, as shown in Example 12.

Example 12 Prototypes of the member STDAPIINFO

```
D CreateUserSpace...
     D                 PR                  ExtPgm('QUSCRTUS')
     D  UserSpaceName                20A   Const
     D  Attribute                    10A   Const
     D  Size                         10I 0 Const
     D  Initial                       1A   Const
     D  Authority                    10A   Const
     D  Text                         50A   Const
       // Optional Parameter Group 1
     D  Replace                      10A   Const Options(*NOPASS)
     D  ErrorCode                          Options(*NOPASS)
     D                                     LikeDS(APIError)
       // Optional Parameter Group 2
     D  Domain                       10A   Const Options(*NOPASS)
       // Optional Parameter Group 3
     D  TransferSize                 10I 0 Const Options(*NOPASS)
     D  OptimumAlign                  1A   Const Options(*NOPASS)

     D ChangeUserSpaceAttributes...
     D                 PR                  ExtPgm('QUSCUSAT')
     D  ReturnLibrary                10A
     D  UserSpaceName                20A   Const
     D  Attribute                          Const
     D                                     LikeDS(SpaceAttribute)
     D  ErrorCode                          LikeDS(APIError)

     D GetUserSpace    PR                  ExtPgm('QUSPTRUS')
     D  UserSpaceName                20A   Const
     D  pSpacePtr                      *
     D  ErrorCode                          Options(*NOPASS)
     D                                     LikeDS(APIError)

     D DeleteUserSpace...
     D                 PR                  ExtPgm('QUSDLTUS')
     D  UserSpace                    20A   Const
     D  ErrorCode                          LikeDS(APIError)

     D SpaceAttribute  DS                  Qualified Based(DummyPtr)
     D  NumberOfRecs                 10I 0
     D  ExtendRecord                 12A
     D   Key                         10I 0 Overlay(ExtendRecord)
     D   Length                      10I 0 OverLay(ExtendRecord:*Next)
     D   Extend                       1A   OverLay(ExtendRecord:*Next)
```

The member SPACEPROCS contains the definition of the **CreateListSpace** and **DeleteListSpace** subprocedures, as shown in Example 13 on page 24.

Example 13 CreateListSpace and DeleteListSpace subprocedures in SPACEPROCS

```
H NoMain Option(*SrcStmt : *NoDebugIO)

      /Copy APISRC,STDAPIINFO

     P CreateListSpace...
     P                 B                   Export
     D CreateListSpace...
     D                 PI              *
     D  SpaceName                    10A   Const

     D FullSpaceName   S               20A
     D Library         S               10A
     D SpacePtr        S                 *

     D SetAttribute    DS                  LikeDS(SpaceAttribute)
     D                                     Inz

      /Free
(1)       FullSpaceName = SpaceName + 'QTEMP';
(2)       DeleteUserSpace( FullSpaceName
                         : APIError);
(3)       CreateUserSpace( FullSpacename
                         : 'APILIST'
                         : 1000000
                         : x'00'
                         : '*ALL'
                         : 'User Space for API Output'
                         : '*YES'
                         : APIError);
          SetAttribute.NumberOfRecs = 1;
          SetAttribute.Key = 3;
          SetAttribute.Length = 1;
          SetAttribute.Extend = '1';
(4)       ChangeUserSpaceAttributes( Library
                                   : FullSpaceName
                                   : SetAttribute
                                   : APIError);
(5)       GetUserSpace( FullSpacename
                      : SpacePtr
                      : APIError);

(6)       Return SpacePtr;

      /End-Free
     P                 E


     P DeleteListSpace...
     P                 B                   Export
     D DeleteListSpace...
     D                 PI
     D  SpaceName                    10A   Const

     D FullSpaceName   S               20A
```

```
     /Free
         FullSpaceName = SpaceName + 'QTEMP';
         DeleteUserSpace( FullSpaceName
                        : APIError);
     /End-Free
    P                 E
```

Create the SPACEPROCS service program by using the following commands:

```
CRTRPGMOD MODULE(REDBOOK/SPACEPROCS) SRCFILE(REDBOOK/APISRC)
CRTSRVPGM SRVPGM(REDBOOK/SPACEPROCS) EXPORT(*ALL)
```

Add the service program to the APIS binding directory by using this command:

```
ADDBNDDIRE BNDDIR(REDBOOK/APIS) OBJ((REDBOOK/SPACEPROCS))
```

The main points to note, which are numbered to correspond with the numbering in Example 13 on page 24, are:

1. Only the user space name is provided as a parameter. The subprocedure creates the user space in QTEMP.
2. Even though the user space should not exist in QTEMP, the subprocedure makes sure by deleting the user space.
3. The user space is created with an initial size of 1,000,000 bytes. One of the issues of dealing with the list APIs is that you must ensure that you have enough space for the data returned. One solution is to use the maximum size of 16 M, or the other solution, which you see in Example 13 on page 24, is to make the user space extendable.
4. The attribute of the user space is changed to make it extendable. Refer to section 5.3.1.7 of *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402 for details.
5. The user space is loaded, and the pointer to its memory location is retrieved.
6. The pointer to the user space is returned.

The List API documentation

The documentation for a List API follows a standard format in the Information Center. For any List API you will see the following headings:

- Description
- Authorities and Locks
- Required Parameter Group
- Optional Parameters (if any)
- Format of the Generated List
- Input Parameter Section
- Header Section
- List Data Section (for each possible format request)
- Field Descriptions
- Error Messages

The Format of the Generated List section contains a link for “User Space Format for List APIs.”

User space format for List APIs

Another item that is common to the List APIs is the way in which information is placed in the user space. Although the content of the user space is dependent on the API that is used to populate it, the format in which the information is placed in the user space is standard.

Information is placed in the user space in five segments, as shown in Figure 19. The five segments are:

1. The content of the User Area segment is common to all List APIs, and you can use it as a communication area if you are sharing the user space between multiple programs.
2. The content of the Generic Header segment is common to all List APIs and contains information about the API that is used to populate the user space. Most importantly, it contains pointer offsets to the other segments in the user space, an indication of the number of entries in the list, and the size of each entry.
3. The content of the Input Parameters segment is unique to each API and reflects the values passed as parameters to the API. Therefore, the size of the Input Parameters segment is different for each API.
4. The content of the Header segment is unique for each API and contains general information that is pertinent to the use of the API.
5. The content of the List Data segment is unique for each API and contains the list of data that the API generates.

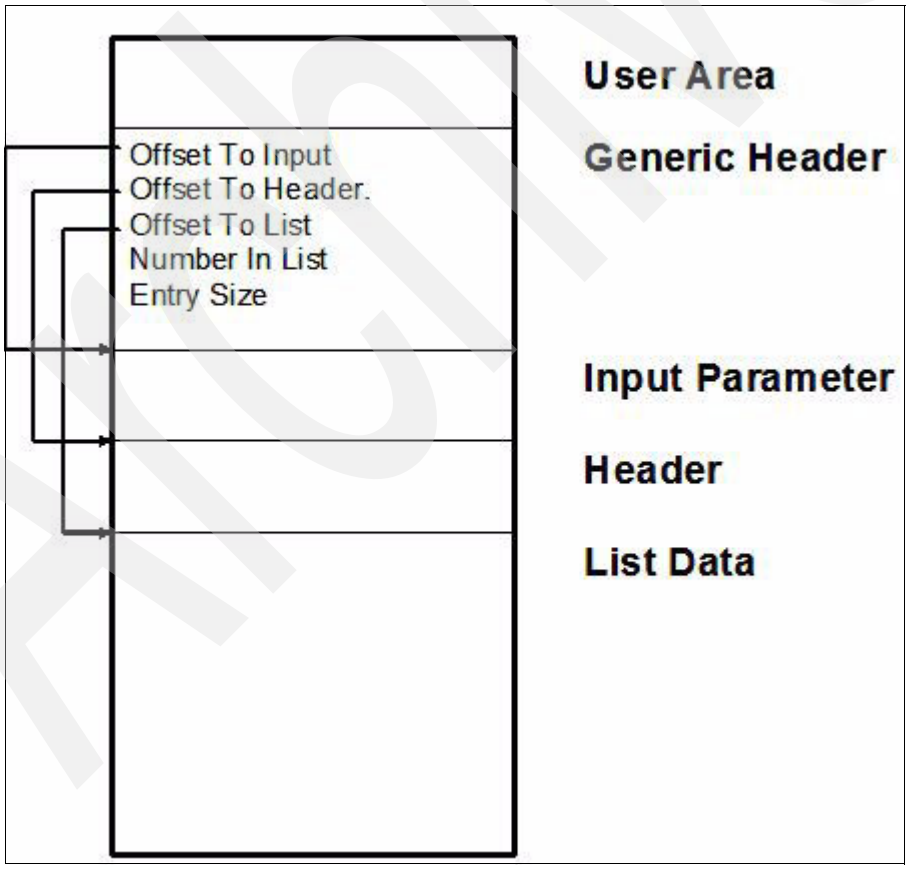


Figure 19 Format of Data in a List API User Space

Figure 20 shows the documentation for the Generic Header.

Common data structure formats

This topic shows the generic user space layout. Format 0100 shows the format for an original program model (OPM) layout. Format 0300 shows the format for an Integrated Language Environment® (ILE) model layout. The fields are described in detail after the table.

Generic header format 0100

Offset		Type	Field
Dec	Hex		
0	0	CHAR(64)	User area
64	40	BINARY(4)	Size of generic header
68	44	CHAR(4)	Structure's release and level
72	48	CHAR(8)	Format name
80	50	CHAR(10)	API used
90	5A	CHAR(13)	Date and time created
103	67	CHAR(1)	Information status
104	68	BINARY(4)	Size of user space used
108	6C	BINARY(4)	Offset to input parameter section
112	70	BINARY(4)	Size of input parameter section
116	74	BINARY(4)	Offset to header section
120	78	BINARY(4)	Size of header section
124	7C	BINARY(4)	Offset to list data section
128	80	BINARY(4)	Size of list data section
132	84	BINARY(4)	Number of list entries
136	88	BINARY(4)	Size of each entry
140	8C	BINARY(4)	CCSID of data in the list entries
144	90	CHAR(2)	Country or region ID
146	92	CHAR(3)	Language ID
149	95	CHAR(1)	Subsetted list indicator
150	96	CHAR(42)	Reserved

Generic header format 0300

Offset		Type	Field
Dec	Hex		
0	0		Everything from the 0100 format
192	C0	CHAR(256)	API entry point name
448	1C0	CHAR(128)	Reserved

Figure 20 Structure of the Generic Header

Example 14 shows that the member STDAPIINFO contains the corresponding definition of the Generic Header.

Example 14 Definition of the Generic Header

```
D Base_GenericHeader...
     D                 DS                  Qualified Based(DummyPtr)
(1)  D  UserArea                     64A
     D  HeaderSize                   10I 0
     D  ReleaseLevel                  4A
     D  Format                        8A
     D  APIUsed                      10A
     D  Created                      13A
     D  Status                        1A
     D  UserSpaceSize                10I 0
(2)  D  OffsetToInput                10I 0
```

```
     D  SizeOfInput                  10I 0
(2)  D  OffsetToHeader...
     D                               10I 0
     D  SizeOfHeader                 10I 0
(2)  D  OffsetToList                 10I 0
     D  SizeOfList                   10I 0
(3)  D  NumberInList                 10I 0
(3)  D  EntrySize                    10I 0
     D  EntryCCSID                   10I 0
     D  CountryID                     2A
     D  LanguageID                    3A
     D  SubsettedList                 1A
     D                               42A
       // Only use for ILE
     D  EntryPointName...
     D                              256A
     D                              128A
```

The Generic Header contains many fields that might or might not be of interest to you, but the following items are the main components of the Generic Header that are of practical use, and they provide the means of accessing the rest of the information in the user space. The following numbered list explains the fields in Example 14 on page 27. The numbers in this list correspond with the numbering in Example 14 on page 27:

1. The UserArea is common to all list APIs and may contain any information you want.
2. **OffsetToInput**, **OffsetToHeader**, and **OffsetToList** provide the offsets to the variable portions of the user space. When the API copies information to the user space, the Header information is always at the start of the user space. You use the address of the user space, plus the relevant offsets, to set the starting locations of the Input Parameters, Header, and List Data sections.
3. **NumberInList** provides the number of entries in the list, and **EntrySize** provides the size of each entry. You use these values to “loop” through the entries in the list.

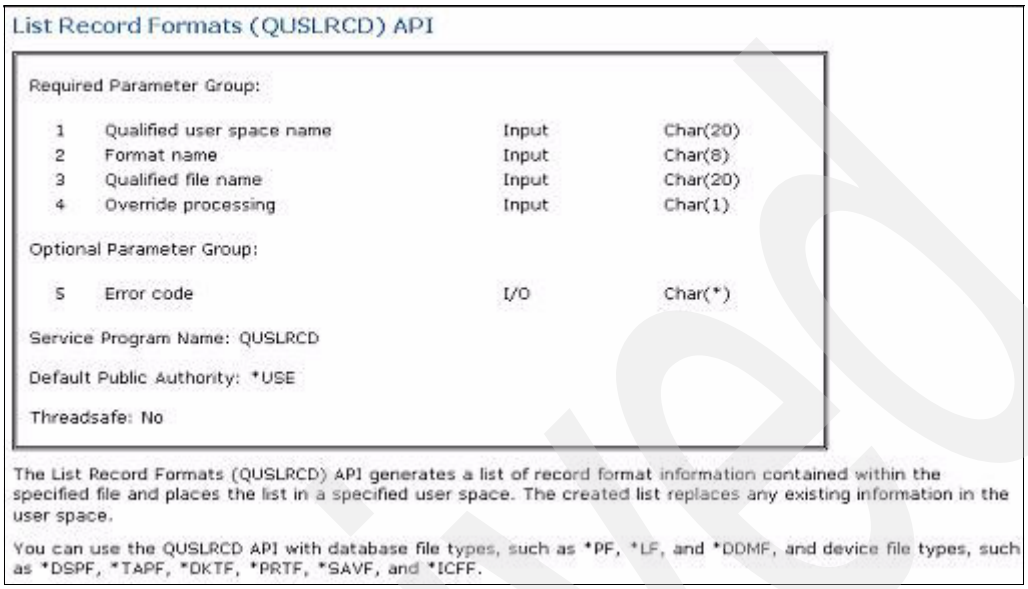
Let us look at an example of how to decipher the contents of the Input Parameters, Header, and List Data segments using the List Record Format (QUSLRCD) and List Fields (QUSLFLD) APIs. These APIs are the foundation for a solution for one of the most commonly asked questions on the RPG Internet lists: How can I obtain the value of a field whose name is stored in another field? After you master the basics of this API, you might want to continue your explorations by studying the code at the Midrange.com FAQ, which you can find at:

http://faq.midrange.com/data/cache/51.html

You can find the descriptions of the QUSLFLD and QUSLRCD APIs in the Information Center using the API finder.

List Record Formats (QUSLRCD)

The *List Record Formats* (QUSLRCD) API lists all record formats for a requested file. Figure 21 shows the documentation for the parameter group and the API description.


List Record Formats (QUSLRCD) API

Required Parameter Group:

1	Qualified user space name	Input	Char(20)
2	Format name	Input	Char(8)
3	Qualified file name	Input	Char(20)
4	Override processing	Input	Char(1)

Optional Parameter Group:

5	Error code	I/O	Char(*)

Service Program Name: QUSLRCD

Default Public Authority: *USE

Threadsafe: No

The List Record Formats (QUSLRCD) API generates a list of record format information contained within the specified file and places the list in a specified user space. The created list replaces any existing information in the user space.

You can use the QUSLRCD API with database file types, such as *PF, *LF, and *DDMF, and device file types, such as *DSPF, *TAPF, *DKTF, *PRTF, *SAVF, and *ICFF.

Figure 21 Parameters for the List Record Formats (QUSLRCD) API

The member STDAPIINFO contains the corresponding prototype.

Example 15 STDAPIINFO corresponding prototype

```
D ListFormats     PR                  ExtPgm('QUSLRCD')
(1)  D  UserSpace                    20A   Const
(2)  D  FormatName                    8A   Const
(3)  D  FileName                     20A   Const
(4)  D  Override                      1A   Const
(5)  D  Error                              LikeDS(APIError)
     D                                     Options(*NoPass)
```

The definitions of the parameters, which correspond with the numbering in Example 15, are:

1. The User Space name identifies the user space in which the list is placed. This twenty-character field identifies the name of the user space in the first ten characters, followed by the name of the library (*CURLIB and *LIBL are allowed for the library name). Ensure that the names are in uppercase.
2. The Format Name identifies the format in which the list information is returned. The allowed values are RCDL0100, RCDL0200, and RCDL0300. This value indicates how to interpret the information that the API returns.
3. The File Name identifies the file you want to examine. The naming rules are the same as the user space name.
4. Override processing indicates whether or not active overrides should be taken into account when the API is called. In other words, if the requested file is overridden to another file, this value indicates whether the API should run against the requested file or the overridden file.
5. The last optional parameter is the standard API error structure.

Figure 22 shows the definition of the Input Parameter Section for the QUSLRCD API.

Input Parameter Section

Offset			
Dec	Hex	Type	Field
0	0	CHAR(10)	User space name
10	A	CHAR(10)	User space library name
20	14	CHAR(8)	Format name
28	1C	CHAR(10)	File name specified
38	26	CHAR(10)	File library name specified
48	30	CHAR(1)	Override processing

Figure 22 Input Parameter section for QUSLRCD

The member STDAPIINFO contains the corresponding data structure definition, as shown in Example 16. The contents of the data structure reflect the parameters specified when QUSLRCD was called.

Example 16 Data structure definition for the Input Parameter section for QUSLRCD

```
D Base_RecordInputParm...
     D                 DS                  Qualified Based(DummyPtr)
     D  UserSpace                    10A
     D  UserSpaceLibrary...
     D                               10A
     D  Format                        8A
     D  File                         10A
     D  FileLibrary                  10A
     D  Override                      1A
```

At first glance, you may think that the Input Parameter Section is superfluous to requirements, but it can be useful if the program or procedure that processes the user space is not the program or procedure that originally called the API.

Figure 23 shows the definition of the Header Section for the QUSLRCD API.

Header Section

Offset			
Dec	Hex	Type	Field
0	0	CHAR(10)	File name used
10	A	CHAR(10)	File library name used
20	14	CHAR(10)	File type
30	1E	CHAR(50)	File text description
80	50	BINARY(4)	File text description CCSID
84	54	CHAR (13)	File creation date

Figure 23 Header Section for QUSLRCD

The member STDAPIINFO contains the corresponding data structure definition, as shown in Example 17.

Example 17 Data structure definition for the Header Section of QUSLRCD

```
D Base_RecordHeader...
     D                 DS                  Qualified Based(DummyPtr)
     D  FileUsed                     10A
     D  FileLibraryUsed...
     D                               10A
     D  FileType                     10A
```

```
     D  FileText                     50A
     D                                1A
     D  FileTextCCSID                10I 0
     D  FileCreationDate...
     D                               13A
```

The Header section provides basic information about the file used. Remember that the actual file that is used might be different from the file requested, depending on the value of the override parameter.

QUSLRCD returns list data in one of three formats, depending on the value of RCDL0100, RCDL0200, or RCDL0300 specified for the Format parameter. Figure 24 shows the definition of the formats for QUSLRCD.

RCDL0100 List Data Section

Offset		Type	Field
Dec	Hex		
0	0	CHAR(10)	Record format name

RCDL0200 List Data Section

Offset		Type	Field
Dec	Hex		
0	0	CHAR(10)	Record format name
10	A	CHAR(13)	Record format ID
23	17	CHAR(1)	Reserved
24	18	BINARY(4)	Record length
28	1C	BINARY(4)	Number of fields
32	20	CHAR(50)	Record text description
82	52	CHAR(2)	Reserved
84	54	BINARY(4)	Record text description CCSID

RCDL0300 List Data Section

Offset		Type	Field
Dec	Hex		
0	0	CHAR(10)	Record format name
10	A	CHAR(2)	Lowest response indicator
12	C	BINARY(4)	Buffer size
16	10	CHAR(20)	Record format type
36	24	CHAR(1)	Starting line number
37	25	CHAR(1)	Separate indicator area present

Figure 24 Formats of List Data for QUSLRCD

The member STDAPIINFO contains the corresponding data structure definitions, as shown in Example 18.

Example 18 Data structure definitions for the Formats of List Data for QUSLRCD

```
D Base_RcdL0100   DS                  Qualified Based(DummyPtr)
     D  FormatName                   10A

     D Base_RcdL0200   DS                  Qualified Based(DummyPtr)
     D  FormatName                   10A
     D  FormatId                     13A
     D                                1A
     D  RecordLength                 10I 0
```

```
D  NumberOfFields...
D                                10I 0
D  FormatText                    50A
D                                 2A
D  FormatTextCCSID...
D                                10I 0

D Base_RcdL0300   DS                   Qualified Based(DummyPtr)
D  FormatName                    10A
D  LowestResponseInd...
D                                 2A
D  BufferSize                    10I 0
D  FormatType                    20A
D  StartLine                      1A
D  INDARAPresent                  1A
D                                 1A
```

The List Data contains an entry in the requested format for each record format that is defined in the file. The RCDL0300 format might only be specified for device files.

List Fields (QUSLFLD)

The QUSLFLD API generates a list of fields for a specified record format in a file. Figure 25 shows the documentation for the parameter group and the API description.

List Fields (QUSLFLD) API

Required Parameter Group:

1	Qualified user space name	Input	Char(20)
2	Format name	Input	Char(8)
3	Qualified file name	Input	Char(20)
4	Record format name	Input	Char(10)
5	Override processing	Input	Char(1)

Optional Parameter:

6	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The List Fields (QUSLFLD) API generates a list of fields within a specified file record format name. The list of fields is placed in a specified user space. The generated list replaces any existing information in the user space. You can use the QUSLFLD API only with database file types, such as *PF, *LF, and *DDMF, and device file types, such as *ICFF and *PRTF.

You can use the QUSLFLD API to:

- Generate a list of field format names.
- Gather additional information about specific field formats.
- Create a product similar to the Structured Query Language (SQL) using the Open Query File (OPNQRYF) command.
- Create applications similar to the data file utility (DFU).
- Create a compiler supporting externally described data.
- Create applications that use data defined to the system.

Figure 25 Parameters for the List Fields (QUSLFLD) API

The member STDAPIINFO contains the corresponding prototype, as shown in Example 19.

Example 19 STDAPIINFO corresponding prototype

```
D ListFields      PR                  ExtPgm('QUSLFLD')
     D  UserSpace                    20A   Const
     D  FormatName                    8A   Const
     D  FileName                     20A   Const
     D  RecordFormat                 10A   Const
     D  Override                      1A   Const
     D  Error                              LikeDS(APIError)
     D                                     Options(*NoPass)
```

The definitions of the parameters are identical to those for QUSLRCD with the addition of the Record Format Name, which identifies the record format in the file for which you want to list the fields. Naturally, the values for the Format Name are different: FLDL0100, FLDL0200, and FLDL0300. The API only retrieves the information for one record format. You can specify a value of *FIRST for the format name (even though this is not indicated in the documentation), which is useful if you are using the API with single format files, but you do not know the name of the format.

Figure 26 shows the definition of the Input Parameter section for the QUSLFLD API.

Input Parameter Section

Offset			
Dec	Hex	Type	Field
0	0	CHAR(10)	User space name
10	A	CHAR(10)	User space library name
20	14	CHAR(8)	Format name
28	1C	CHAR(10)	File name specified
38	26	CHAR(10)	File library name specified
48	30	CHAR(10)	Record format name specified
58	3A	CHAR(1)	Override processing

Figure 26 Input Parameter section for QUSLFLD

The member STDAPIINFO contains the corresponding data structure definition, as shown in Example 20. The contents of the data structure reflect the parameters specified when QUSLFLD was called.

Example 20 Data structure definition for QUSLFLD Input Parameter section

```
     D Base_FieldInputParm...
     D                 DS                  Qualified Based(DummyPtr)
     D  UserSpace                    10A
     D  UserSpaceLibrary...
     D                               10A
     D  Format                        8A
     D  File                         10A
     D  FileLibrary                  10A
     D  FileFormat                   10A
     D  Override                      1A
```

Figure 27 shows the definition of the Header section for the QUSLFLD API.

Header Section

Offset		Type	Field
Dec	Hex		
0	0	CHAR(10)	File name used
10	A	CHAR(10)	File library name used
20	14	CHAR(10)	File type
30	1E	CHAR(10)	Record format name used
40	28	BINARY(4)	Record length
44	2C	CHAR(13)	Record format ID
57	39	CHAR(50)	Record text description
107	6B	CHAR(1)	Reserved
108	6C	BINARY(4)	Record text description CCSID
112	70	CHAR(1)	Variable length fields in format indicator
113	71	CHAR(1)	Graphic fields indicator
114	72	CHAR(1)	Date and time fields indicator
115	73	CHAR(1)	Null-capable fields indicator

Figure 27 Header Section for QUSLFLD

The member STDAPIINFO contains the corresponding data structure definition, as shown in Example 21.

Example 21 Data structure definition for the Header section for QUSLFLD

```
D Base_FieldHeader...
     D                 DS                  Qualified Based(DummyPtr)
     D  FileUsed                     10A
     D  FileLibraryUsed...
     D                               10A
     D  FileType                     10A
     D  FormatUsed                   10A
     D  RecordLength                 10I 0
     D  FormatId                     13A
     D  FormatText                   50A
     D                                1A
     D  FormatTextCCSID...
     D                               10I 0
     D  VariableLengthFieldsInd...
     D                                1A
     D  GraphicFieldsInd...
     D                                1A
     D  DateTimeFieldsInd...
     D                                1A
     D  NullCapableFieldsInd...
     D                                1A
```

The Header section provides basic information about the record format that is being listed, such as the record length and format ID. It also contains indicators (zero or one) as to whether or not the record contains Graphic Fields, Date/Time Fields, or Null Capable Fields.

QUSLFLD returns list data in one of three formats, depending on the value of FLDL0100, FLDL0200, or FLDL0300 specified for the Format parameter.

Figure 28 shows the definition of the FLDL0100 format for QUSLFLD.

FLDL0100 List Data Section

Offset		Type	Field
Dec	Hex		
0	0	CHAR(10)	Field name
10	A	CHAR(1)	Data type
11	B	CHAR(1)	Use
12	C	BINARY(4)	Output buffer position
16	10	BINARY(4)	Input buffer position
20	14	BINARY(4)	Field length in bytes
24	18	BINARY(4)	Digits
28	1C	BINARY(4)	Decimal position
32	20	CHAR(50)	Field text description
82	52	CHAR(2)	Edit code
84	54	BINARY(4)	Edit word length
88	58	CHAR(64)	Edit word
152	98	CHAR(20)	Column heading 1
172	AC	CHAR(20)	Column heading 2
192	C0	CHAR(20)	Column heading 3
212	D4	CHAR(10)	Internal field name
222	DE	CHAR(30)	Alternative field name
252	FC	BINARY(4)	Length of alternative field name
256	100	BINARY(4)	Number of DBCS characters
260	104	CHAR(1)	Null values allowed
261	105	CHAR(1)	Host variable indicator
262	106	CHAR(4)	Date and time format
266	10A	CHAR(1)	Date and time separator
267	10B	CHAR(1)	Variable length field indicator (overlay for MI mapping)
268	10C	BINARY(4)	Field text description CCSID
272	110	BINARY(4)	Field data CCSID
276	114	BINARY(4)	Field column headings CCSID
280	118	BINARY(4)	Field edit words CCSID
284	11C	BINARY(4)	UCS-2 displayed field length
288	120	BINARY(4)	Field data encoding scheme
292	124	BINARY(4)	Maximum large object field length
296	128	BINARY(4)	Pad length for large object
300	12C	BINARY(4)	Length of user-defined type name
304	130	CHAR(128)	User-defined type name
432	1B0	CHAR(10)	User-defined type library name
442	1BA	CHAR(1)	Datalink link control
443	1BB	CHAR(1)	Datalink integrity
444	1BC	CHAR(1)	Datalink read permission
445	1BD	CHAR(1)	Datalink write permission
446	1BE	CHAR(1)	Datalink recovery
447	1BF	CHAR(1)	Datalink unlink control
448	1C0	BINARY(4)	Display or print row number
452	1C4	BINARY(4)	Display or print column number
456	1C8	CHAR(1)	ROWID column
457	1C9	CHAR(1)	Identity column
458	1CA	CHAR(1)	GENERATED BY
459	1CB	CHAR(1)	Identity column - CYCLE
460	1CC	DECIMAL(31,0)	Identity column - Original START WITH
476	1DC	DECIMAL(31,0)	Identity column - Current START WITH
492	1EC	BINARY(4)	Identity column - INCREMENT BY
496	1F0	DECIMAL(31,0)	Identity column - MINVALUE
512	200	DECIMAL(31,0)	Identity column - MAXVALUE
528	210	BINARY(4)	Identity column - CACHE
532	214	CHAR(1)	Identity column - ORDER
533	215	CHAR(11)	Reserved

Figure 28 FLDL0100 Format returned for QUSLFLD

The member STDAPIINFO contains the corresponding data structure definition, as shown in Example 22.

Example 22 Data structure definition for FLDL0100 for QUSFLD

```
D Base_FldL0100   DS                  Qualified Based(DummyPtr)
     D  FieldName                    10A
     D  DataType                      1A
     D  Use                           1A
     D  OutputBufferPosition...
     D                               10I 0
     D  InputBufferPosition...
     D                               10I 0
     D  FieldLengthBytes...
     D                               10I 0
     D  Digits                       10I 0
     D  Decimals                     10I 0
     D  Text                         50A
     D  EditCode                      2A
     D  EditWordLength...
     D                               10I 0
     D  EditWord                     64A
     D  ColumnHeading1...
     D                               20A
     D  ColumnHeading2...
     D                               20A
     D  ColumnHeading3...
     D                               20A
     D  InternalFieldname...
     D                               10A
     D  AlternativeFieldname...
     D                               30A
     D  AlternativeFieldnameLength...
     D                               10I 0
     D  NumberOfDBCS                 10I 0
     D  AllowNULL                     1A
     D  HostVariableInd...
     D                                1A
     D  DateTimeFormat...
     D                                4A
     D  DateTimeSeparator...
     D                                1A
     D  VarLenFieldInd...
     D                                1A
     D  TextCCSID                    10I 0
     D  DataCCSID                    10I 0
     D  HeadingCCSID                 10I 0
     D  EditWordCCSID                10I 0
     D  UCS2Length                   10I 0
     D  EncodingScheme...
     D                               10I 0
     D  MaxLOBLength                 10I 0
     D  PadLengthLOB                 10I 0
     D  UDTNameLength                10I 0
     D  UDTName                     128A
     D  UDTLibrary                   10A
```

```
D  DatalinkLinkControl...
D                                 1A
D  DatalinkIntegrity...
D                                 1A
D  DatalinkReadPermission...
D                                 1A
D  DatalinkWritePermission...
D                                 1A
D  DatalinkRecovery...
D                                 1A
D  DatalinkUnlinkControl...
D                                 1A
D  RowNumber                     10I 0
D  ColumnNumber                  10I 0
D  ROWIDColumn                    1A
D  IdentityColumn...
D                                 1A
D  GeneratedBy                    1A
D  IdentityColumn_Cycle...
D                                 1A
D  IdentityColumn_OriginalStartWith...
D                                31P 0
D  IdentityColumn_CurrentStartWith...
D                                31P 0
D  IdentityColumn_IncrementBy...
D                                10I 0
D  IdentityColumn_MinValue...
D                                31P 0
D  IdentityColumn_MaxValue...
D                                31P 0
D  IdentityColumn_Cache...
D                                10I 0
D  IdentityColumn_Order...
D                                 1A
D                                11A
```

Now, all you have to decide is which items of information are relevant. It usually takes some experimentation to interpret the information returned. For example, FLDL0100 returns the number of bytes of storage required for a field, not the length of the field. If the field is numeric, you can use the digits for the length. If the field is a varying length character field, subtract two from the storage length.

Figure 29 shows the definition of the FLDL0200 and FLDL0300 formats.

FLDL0200 List Data Section

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Length of FLDL0200 format
4	4	BINARY(4)	Displacement to default value
8	8	BINARY(4)	Length of default value
12	C		All fields defined by FLDL0100 format
*	*	CHAR(*)	Default value

FLDL0300 List Data Section

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Length of FLDL0300 format
4	4	BINARY(4)	Displacement to all fields defined by FLDL0100 format
8	8	BINARY(4)	Displacement to alternative field name
12	C	BINARY(4)	Displacement to default value
16	10	BINARY(4)	Length of default value
*	*		All fields defined by FLDL0100 format
*	*	CHAR(*)	Alternative field name (long)
*	*	CHAR(*)	Default value

Figure 29 FLDL0200 and FLDL0300 Formats returned for QUSLFLD

The member STDAPIINFO contains the corresponding data structure definitions, as shown in Example 23.

Example 23 Data structure definitions for FLDL0200 and FLDL0300 for QUSLFLD

```
D Base_FldL0200   DS                  Qualified Based(DummyPtr)
D  FldL0200Length...
D                                10I 0
D  DefaultOffset                 10I 0
D  DefaultLength                 10I 0
  //  FldL0100                          LikeDS(Base_FldL0100)
  //  DefaultValue                   A

D Base_FldL0300   DS                   Qualified Based(DummyPtr)
D  FldL0300Length...
D                                10I 0
D  FldL0100Offset...
D                                10I 0
D  AlternativeOffset...
D                                10I 0
D  DefaultOffset                 10I 0
D  DefaultLength                 10I 0
  //  FldL0100                          LikeDS(Base_FldL0100)
  //  AlternativeName                A
  //  DefaultValue                   A
```

FLDL0200 returns the exact same information that FLDL0100 returns along with the default value for the field. FLDL0300 returns the exact same information as FLDL0100 along with the alternative field name and the default value for the field. The alternative field name and the default value for the field are variable in length, so you must use the offsets that are provided in the formats along with a pointer to map fields onto the relevant space.

Because FLDL0200 and FLDL0300 are variable in length, there is an implication in using them as opposed to FLDL0100, which is fixed length. Instead of using the Size of Each Entry in the Generic Header to determine the starting point of the next entry, you must use the format length (FldL0200Length and FldL0300Length), for example, the next list entry is at the address of this entry plus the length.

Using QUSLRCD and QUSLFLD

The program API03 is an example of a program that uses the QUSLRCD and QUSFLD API to list record format and field information for a file, as shown in Figure 30. You enter the name of a file and the library. A list of record formats in the file is displayed. You select a format, and a list of fields in the file is displayed.

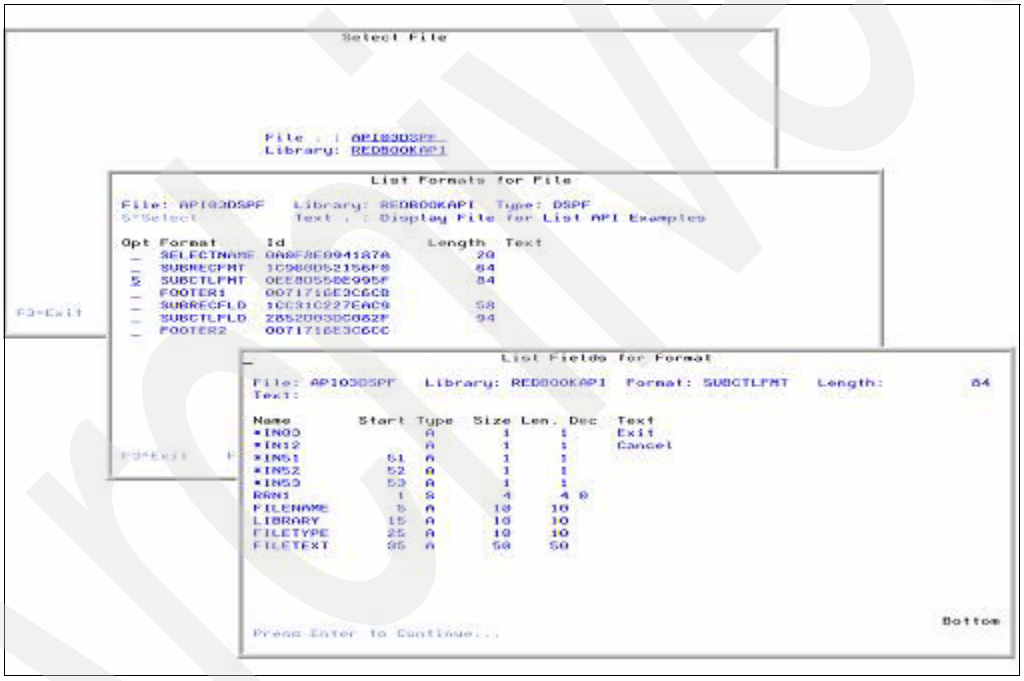


Figure 30 Program using the QUSLRCD and QUSLFLD APIs

The member API03DSPF contains the definition of the display file, as shown in Example 24.

Example 24 Definition of the display file for QUSLREC and QUSLFLD List API example

```
     A* API03D - Display File for QUSLREC and QUSLFLD List API Example

     A                                      DSPSIZ(24 80 *DS3)
     A                                      INDARA

(1)  A          R SELECTNAME
     A                                      CA03(03 'Exit')
     A                                  1 36'Select File'
```

```
     A                                      DSPATR(HI)
     A                                   9 28'File . :'
     A            FILENAME      10   B  9 37
     A  31                                  ERRMSG('Invalid File or Library')
     A                                  10 28'Library:'
     A            LIBRARY       10   B 10 37
     A                                  23  2'F3=Exit'
     A                                      DSPATR(HI)
     A                                      COLOR(BLU)

(2)  A          R SUBRECFMT
     A                                      SFL
     A            OPTION         1   B  7  3
     A            FORMATNAME    10   0  7  6
     A            FORMATID      13   0  7 17
     A            LENGTH        10  0O  7 31EDTCDE(Z)
     A            TEXT          30   0  7 42

(2)  A          R SUBCTLFMT
     A                                      SFLCTL(SUBRECFMT)
     A                                      OVERLAY
     A                                      CA03(03 'Exit')
     A                                      CA12(12 'Cancel')
     A  51                                  SFLDSP
     A  52                                  SFLDSPCTL
     A  53                                  SFLCLR
     A  51                                  SFLEND(*MORE)
     A                                      SFLSIZ(1000)
     A                                      SFLPAG(15)
     A            RRN1           4S 0H
     A                                   1 28'List Formats for File'
     A                                      DSPATR(HI)
     A                                   3  2'File:'
     A            FILENAME      10   0  3  8
     A                                   3 20'Library:'
     A            LIBRARY       10   0  3 29
     A                                   3 41'Type:'
     A            FILETYPE      10   0  3 47
     A                                   4  2'5=Select'
     A                                      COLOR(BLU)
     A                                   4 20'Text . :'
     A            FILETEXT      50   0  4 29
     A                                   6  2'Opt'
     A                                      DSPATR(HI)
     A                                   6  6'Format'
     A                                      DSPATR(HI)
     A                                   6 17'Id'
     A                                      DSPATR(HI)
     A                                   6 34'Length'
     A                                      DSPATR(HI)
     A                                   6 42'Text'
     A                                      DSPATR(HI)

(2)  A          R FOOTER1
     A                                  23  2'F3=Exit    F12=Cancel'
```

```
     A                                      DSPATR(HI)
     A                                      COLOR(BLU)

(3)  A          R SUBRECFLD
     A                                      SFL
     A            FLDNAME       10   0  7  2
     A            OFFSET         5  00  7 13EDTCDE(Z)
     A            FLDTYPE        1   0  7 20
     A            FLDSIZE        5  00  7 24EDTCDE(Z)
     A            FLDLENGTH      5  00  7 30EDTCDE(Z)
     A            DECPOS         2   0  7 36
     A            FLDTEXT       30   0  7 40

(3)  A          R SUBCTLFLD
     A                                      SFLCTL(SUBRECFLD)
     A                                      OVERLAY
     A  51                                  SFLDSP
     A  52                                  SFLDSPCTL
     A  53                                  SFLCLR
     A  51                                  SFLEND(*MORE)
     A                                      SFLSIZ(1000)
     A                                      SFLPAG(15)
     A            RRN2           4S 0H
     A                                    1 28'List Fields for Format'
     A                                      DSPATR(HI)
     A                                    3  2'File:'
     A            FILENAME      10   0  3  8
     A                                    3 20'Library:'
     A            LIBRARY       10   0  3 29
     A                                    3 41'Format:'
     A            FORMAT        10   0  3 49
     A                                    3 61'Length:'
     A            RCDLENGTH     10  00  3 69EDTCDE(3)
     A                                    4  2'Text:'
     A            FMTTEXT       50   0  4  8
     A                                    6  2'Name'
     A                                      DSPATR(HI)
     A                                    6 13'Start'
     A                                      DSPATR(HI)
     A                                    6 19'Type'
     A                                      DSPATR(HI)
     A                                    6 25'Size'
     A                                      DSPATR(HI)
     A                                    6 30'Len.'
     A                                      DSPATR(HI)
     A                                    6 35'Dec'
     A                                      DSPATR(HI)
     A                                    6 40'Text'
     A                                      DSPATR(HI)

(3)  A          R FOOTER2
     A                                   23  2'Press Enter to Continue...'
     A                                      DSPATR(HI)
     A                                      COLOR(BLU)
```

Create the display file, in Example 24 on page 39, by using the command:

```
CRTDSPFF FILE(REDBOOK/API03DSPF) SRCFILE(REDBOOK/APISRC)
```

The formats in the display file, which are numbered to correspond with the numbering in Example 24 on page 39, are:

1. SELECTNAME allows for the entry of a file name and a library name. An error message is displayed if the file is not found.
2. SUBRECFMT, SUBCTLFMT, and FOOTER1 display the list of record formats and allow for the selection of a format.
3. SUBRECFLD, SUBCTLFLD, and FOOTER2 display the list of fields for the selected format.

Also note, that we used the file level INDARA keyword to enable the use of the INDDS keyword in the RPGLE program.

Example 25 shows the H, F, and D specs in the API03 program.

Example 25 H, F, and D specs in the AP103 program

```
(1)  H DftActGrp(*No) ActGrp(*New) BndDir('APIS')
     H Option(*SrcStmt:*NoDebugIO)

(2)  FAPI03DSPF CF   E            WorkStn IndDs(WorkStnInd)
     F                                    SFile(SubRecFmt: RRN1)
     F                                    SFile(SubRecFld: RRN2)

       // Prototypes for Internal Procedures
(3)  D CreateSpaces    PR
     D DeleteSpaces    PR
     D SelectFile      PR
     D LoadFormats     PR
     D  pFormatHeader                *   Const
     D ShowFormats     PR
     D ProcessFormats  PR
     D ListFieldsForFormat...
     D                 PR
     D  RecordFormat                10A   Const
     D  pFieldHeader                 *   Const

      /Copy APISRC,STDAPIINFO

(4)   /Copy APISRC,WORKSTNIND
(5)  D  BadName                      N   OverLay(WorkStnInd:31)

     D CtlEvent        S             4A
(6)  D pFormatHeader   S              *
(7)  D pFieldHeader    S              *
(8)  D FormatSpace     S             10A   Inz('LISTFORMAT')
     D FieldSpace      S             10A   Inz('LISTFIELDS')
```

The main points, which we numbered to correspond with the numbering in Example 25 on page 42, are:

1. The APIS binding directory contains an entry for the SPACEPROCS service program, which contains the user space subprocedures that we described in “User spaces” on page 22.
2. The INDDS keyword is used to map the display file indicators to a data structure as opposed to the RPG indicators.
3. Prototypes are defined for the internal subprocedures in the program.
4. A copy member is used to include the standard indicators that are used with a display file.
5. The error indicator **BadName** is appended to the end of the data structure.
6. **FormatHeader** is the pointer used to access the user space that is populated by QUSLRCD.
7. **FieldHeader** is the pointer used to access the user space that is populated by QUSLFLD.
8. **FormatSpace** and **FieldSpace** contain the names of the user spaces.

Example 26 shows the mainline in the API03 program.

Example 26 The mainline in the AP103 program

```
/Free
(1)       CreateSpaces();
          CtLEvent = 'SELF';

(2)       DoU CtlEvent = '*END';
             Select;
                When CtlEvent = 'SELF';
(3)                SelectFile();
                When CtlEvent = 'LOAD';
(4)                LoadFormats(pFormatHeader);
                When CtlEvent = 'SHOW';
(5)                ShowFormats();
                When CtlEvent = 'PROC';
(6)                ProcessFormats();
             EndSl;
          EndDo;

(7)       DeleteSpaces();
          *InLR = *On;
       /End-Free
```

The AP103 program works as follows (the following numbers correspond with the numbering in Example 26):

1. Create the two user spaces.
2. The **Do Until** loop is based on the contents of **CtlEvent**. Again, based on the contents of **CtlEvent**, one subprocedure is called on each iteration of the loop.
3. Select the file and library.

4. Load the list of formats to the subfile. The pointer to the Record Formats user space is passed as a parameter.
5. Display and input the Record Formats subfile.
6. Process requests from the Records Formats subfile.
7. Delete the user spaces.

Next, we look at the internal subprocedures.

CreateSpaces() and DeleteSpaces()

The **CreateSpaces()** and **DeleteSpaces()** subprocedures simply call the standard procedures in SPACEPROCS to create/delete the required user spaces, as shown in Example 27.

Example 27 CreateSpaces() and DeleteSpaces()

```
//-------------------------------------------------------------
       // Create User Spaces used by APIs
     P CreateSpaces    B
     D                 PI
      /Free
          pFormatHeader = CreateListSpace(FormatSpace);
          pFieldHeader = CreateListSpace(FieldSpace);
      /End-Free
     P                 E


       //-------------------------------------------------------------
       // Delete User Spaces used by APIs
     P DeleteSpaces    B
     D                 PI
      /Free
          DeleteListSpace(FormatSpace);
          DeleteListSpace(FieldSpace);
      /End-Free
     P                 E
```

SelectFile()

The **SelectFile()** subprocedure allows for the display and entry of the file selection window, as shown in Example 28. The next event is to load the Record Format subfile and an exit event is set if F3 is pressed.

Example 28 SelectFile()

```
//-------------------------------------------------------------
       // Select the file to process
     p SelectFile      B
     D                 PI
      /Free
         CtlEvent = 'LOAD';

         ExFmt SelectName;
         ErrInds = *Zeros;

         Select;
            When F3Exit;
```

```
                CtlEvent = '*END';
           EndSl;
        /End-Free
     p                 E
```

LoadFormats()

The **LoadFormats()** subprocedure calls QUSLRCD to populate the Record Format user space and loads the list of formats to the subfile. Example 29 shows the D specs for **LoadFormats()**.

Example 29 LoadFormats()

```
//-------------------------------------------------------------
       // Load the list of Formats to a subfile
     p LoadFormats     B
     D                 PI
(1)  D  pFormatHeader                  *   Const

       // General API List Header
(2)  D GenericHeader   DS                  LikeDS(Base_GenericHeader)
     D                                     Based(pFormatHeader)

        // QUSLRCD Formats Returned
     D pInputParameter...
     D                 S                 *
(3)  D InputParameter  DS                  LikeDS(Base_RecordInputParm)
     D                                     Based(pInputParameter)

     D pHeader         S                 *
(4)  D Header          DS                  LikeDS(Base_RecordHeader)
     D                                     Based(pHeader)

     D pRcdL0200       S                 *
(5)  D RcdL0200        DS                  LikeDS(Base_RcdL0200)
     D                                     Based(pRcdL0200)
```

The items to note, which are numbered to correspond with the numbers in Example 29, in the D specs are:

1. The pointer to the Record Format user space is passed as a parameter.
2. LIKEDS defines a Generic Header data structure, which is the same as the standard definition in STDAPIINFO. The data structure is based on the value of the pointer that is passed as a parameter (**pFormatHeader**), which means that the data structure overlays the start of the Record Format user space.
3. LIKEDS defines an Input Parameter data structure, which is the same as the standard definition in STDAPIINFO. The data structure is based on the value of the pointer because it must be overlaid onto the Record Format user space based on the offset in the Generic Header.

4. LIKEDS defines a Header data structure, which is the same as the standard definition in STDAPIINFO. The data structure is based on the value of the pointer because it must be overlaid onto the Record Format user space based on the offset in the Generic Header.
5. LIKEDS defines a RCDL0200 data structure, which is the same as the standard definition in STDAPIINFO. The data structure is based on the value of the pointer because it must be overlaid onto the Record Format user space based on the offset in the Generic Header. This data structure is re-overlaid in the user space for each entry in the list.

Example 30 shows the process in **LoadFormats()**.

Example 30 The process in LoadFormats()

```
      /Free
(1)      CtlEvent = 'SHOW';

(2)      ListFormats( FormatSpace + 'QTEMP'
                    : 'RCDL0200'
                    : FileName + Library
                    : '0'
                    : APIError);

(3)      If APIError.BytesAvail > 0;
             BadName = *On;
             CtlEvent = 'SELF';
             Return;
         EndIf;

(4)      pInputParameter = pFormatHeader + GenericHeader.OffsetToInput;
(5)      pHeader = pFormatHeader + GenericHeader.OffsetToHeader;
(6)      pRcdL0200 = pFormatHeader + GenericHeader.OffsetToList;

(7)      FileType = Header.FileType;
         FileText = Header.FileText;
         Option = *Blanks;

         SFLClr = *On;
         Write SUbCtlFmt;
         SFLClr = *Off;

(8)      For RRN1 = 1 to GenericHeader.NumberInList;
(9)         FormatName = RcdL0200.FormatName;
            FormatId = RcdL0200.FormatId;
            Length = RcdL0200.RecordLength;
            Text = RcdL0200.FormatText;
            Write SubRecFmt;
(10)        pRcdL0200 = pRcdL0200 + GenericHeader.EntrySize;
         EndFor;
       /End-Free
     P                 E
```

The following is an explanation of the process in Example 30 on page 46. The numbers in the following list correspond with the numbers in Example 30 on page 46:

1. The next event is to display the Record Formats subfile.
2. Call QUSLRCD to populate the Record Formats user space with information for the requested file in the RCDL0200 format.
3. If there is an error on the call to QUSLRCD, display the error message on the file selection window. Here, the assumption is that the file or library does not exist, but you can be as detailed as you want to be on the error reporting. We recommend that you check if an error is reported after making a call to a List API because there are a myriad of errors that can occur (from typos to authority problems). Remember, you can find a list of possible error messages at the end of the description of each API. You can also check the value of the status field in the Generic Header (Status) to determine whether or not you have a complete list, which is indicated by a value of C.
4. Overlay the Input Parameter data structure onto the user space, based on the address of the user space plus the offset in the Generic Header.
5. Overlay the Header data structure onto the user space based on the address of the user space plus the offset in the Generic Header.
6. Overlay the RCDL0200 data structure onto the user space based on the address of the user space, plus the offset in the Generic Header. RCDL0200 now overlays the first entry in the list.
7. Set the window headings, and clear the subfile.
8. Use the number of entries in the list (as indicated in the Generic Header) to determine how many records to load to the subfile (RRN1 is the relative-record number field for the Record Format subfile).
9. Copy the required fields from the RCDL0200 data structure to the subfile record, and write the subfile record.
10. Advance the RCDL0200 data structure to the next entry in the list by adding the Entry Size to the current pointer for RCDL0200.

ShowFormats()

The **ShowFormats()** subprocedure allows for the display and entry of the Record Formats subfile. The next event is to process the Record Format subfile. An exit event is set if F3 is pressed and a Select File event is set if F12 is pressed. See Example 31.

Example 31 ShowFormats()

```
//-------------------------------------------------------------
      // Show the Formats for a File
    p ShowFormats     B
    D                 PI
     /Free
        CtlEvent = 'PROC';

        SFLDsp = *On;
        SFLDspCtl = *On;
        Write Footer1;
        ExFmt SubCtlFmt;
        SFLDsp = *Off;
        SFLDspCtl = *Off;

        Select;
```

```
            When F3Exit;
               CtlEvent = '*END';
            When F12Cancel;
               CtlEvent = 'SELF';
         EndSl;

      /End-Free
     P                 E
```

ProcessFormats()

The **ProcessFormats()** subprocedure processes requests in the Record Formats subfile, as shown in Example 32. For any changed record with an option of 5 the **ListFieldsForFormat()** subprocedure is called passing the requested Record Format name and the pointer to the Fields List user space.

Example 32 ProcessFormats()

```
       //-------------------------------------------------------------
       // Process the Formats for a File
     p ProcessFormats  B
     D                 PI
      /Free
         CtlEvent = 'SHOW';

         RRN1 = 0;
         ReadC SubRecFmt;
         Dow Not %EOF();
            If Option = '5';
               ListFieldsForFormat(FormatName:pFieldHeader);
            EndIf;
            Option = *Blanks;
            Update SubRecFmt;
            ReadC SubRecFmt;
         EndDo;

      /End-Free
     P                 E
```

ListFieldsForFormat()

The **ListFieldsForFormat()** bears a striking similarity to the **LoadFormats()** subprocedure, and it should because both subprocedures perform similar functions. **ListFieldsForFormat()** calls QUSLFLD to populate the Fields List user space, loads the list of fields to the subfile, and displays the subfile. Example 33 shows the D specs for **ListFieldsForFormat()**.

Example 33 D specs for ListFieldsForFormat()

```
//-------------------------------------------------------------
       // List the the Fields for a Format
     P ListFieldsForFormat...
     p                 B
     D ListFieldsForFormat...
     D                 PI
```

```
(1)  D  RecordFormat                10A   Const
(2)  D  pFieldHeader                 *    Const

       // General API List Header
(3)  D GenericHeader   DS                 LikeDS(Base_GenericHeader)
     D                                    Based(pFieldHeader)

        // QUSLFLD Formats Returned
     D pInputParameter...
     D                 S               *
(4)  D InputParameter  DS                 LikeDS(Base_FieldInputParm)
     D                                    Based(pInputParameter)

     D pHeader         S               *
(5)  D Header          DS                 LikeDS(Base_FieldHeader)
     D                                    Based(pHeader)

     D pFldL0100       S               *
(6)  D FldL0100        DS                 LikeDS(Base_FldL0100)
     D                                    Based(pFldL0100)
```

The following list highlights the items to note in the D specs. The items in the list are numbered in correspondence to the numbers in Example 33 on page 48:

1. The name of the required record format is passed as a parameter.
2. The pointer to the Field List user space is passed as a parameter.
3. LIKEDS defines a Generic Header data structure, which is the same as the standard definition in STDAPIINFO. The data structure is based on the value of the pointer that is passed as a parameter (**pFieldHeader**), which means that the data structure overlays the start of the Field List user space.
4. LIKEDS defines an Input Parameter data structure that is the same as the standard definition in STDAPIINFO. The data structure is based on the value of the pointer because it must be overlaid onto the Field List user space that is based on the offset in the Generic Header.
5. LIKEDS defines a Header data structure that is the same as the standard definition in STDAPIINFO. The data structure is based on the value of the pointer because it must be overlaid onto the Field List user space that is based on the offset in the Generic Header.
6. LIKEDS defines a FLDL0100 data structure, which is the same as the standard definition in STDAPIINFO. The data structure is based on the value of the pointer because it must be overlaid onto the Field List user space that is based on the offset in the Generic Header. This data structure is re-overlaid in the user space for each entry in the list.

Example 34 shows the process in **ListFieldsForFormat ()**.

Example 34 The process in ListFieldsForFormat()

```
/Free
(1)       ListFields( FieldSpace + 'QTEMP'
                    : 'FLDL0100'
                    : FileName + Library
                    : RecordFormat
                    : '0'
                    : APIError);
```

```
(2)        If APIError.BytesAvail > 0;
              Dsply APIError.MsgId;
              Return;
           EndIf;

(3)        pInputParameter = pFieldHeader + GenericHeader.OffsetToInput;
(4)        pHeader = pFieldHeader + GenericHeader.OffsetToHeader;
(5)        pFldLO100 = pFieldHeader + GenericHeader.OffsetToList;

(6)        Format = Header.FormatUsed;
           RcdLength = Header.RecordLength;
           FmtText = Header.FormatText;

           SFLClr = *On;
           Write SUbCtlFld;
           SFLClr = *Off;

(7)        For RRN2 = 1 to GenericHeader.NumberInList;

(8)           FldName = FldLO100.FieldName;
              OffSet = FldLO100.OutputBufferPosition;
              FldType = FldLO100.DataType;
              FldSize = FldLO100.FieldLengthBytes;
              If (FldLO100.Digits = 0);
                 FldLength = FldLO100.FieldLengthBytes;
                 DecPos = *Blanks;
                 If (FldLO100.VarLenFieldInd = '1');
                   FldLength = FldLength - 2;
                 EndIf;
              Else;
                 FldLength = FldLO100.Digits;
                 DecPos = %Char(FldLO100.Decimals);
              EndIf;
              FldText = FldLO100.Text;
              Write SubRecFld;

(9)           pFldLO100 = pFldLO100 + GenericHeader.EntrySize;
           EndFor;

(10)       SFLDsp = *On;
           SFLDspCtl = *On;
           Write Footer2;
           ExFmt SubCtlFld;
           SFLDsp = *Off;
           SFLDspCtl = *Off;

       /End-Free
     P                 E
```

Create the **ListFieldsForFormat()** program by using the command:

```
CRTBNDRPG PGM(REDBOOK/API03) SRCFILE(REDBOOK/APISRC)
```

The following list explains the process of creating the **`ListFieldsForFormat()`** program. The numbers in this list correspond with the numbers in Example 34 on page 49:

1. Call QUSLFLD to populate the Fields List user space with information for the requested file/record format in the FLDL0100 format.
2. There really must not be any error on the call to QUSLFLD (any errors should have been detected on the call to QUSLRCD, and the record format name should be valid, since it was selected from the displayed list). However, if there is an error, simply display the error message ID and return.
3. Overlay the Input Parameter data structure onto the user space based on the address of the user space, plus the offset in the Generic Header.
4. Overlay the Header data structure onto the user space based on the address of the user space, plus the offset in the Generic Header.
5. Overlay the FLDL0100 data structure onto the user space based on the address of the user space plus the offset in the Generic Header. FLDL0100 now overlays the first entry in the list.
6. Set the window headings, and clear the subfile.
7. Use the number of entries in the list (as indicated in the Generic Header) to determine how many records to load to the subfile (RRN2 is the relative-record number field for the Fields List subfile).
8. Copy the required fields from the FLDL0100 data structure to the subfile record, and write the subfile record. Note how the field length is calculated based on whether it is numeric and whether or not it is varying length.
9. Advance the FLDL0100 data structure to the next entry in the list by adding the Entry Size to the current pointer for FLDL0100. If you were using the FLDL0200 or FLDL0300 formats, you would add the entry size length in the FLDL0200 or FLDL0300 format to the current pointer for the list entry.
10. Display the subfile.

Points to ponder about List APIs

One of the main features of the example in API03 is the way in which the user spaces are reused. The two user spaces are created when the program starts. Each time a file is selected, the Record Formats user space is repopulated. Each time a record format is selected the Fields List user space is repopulated.

Remember:

- List APIs place information in a user space.
- The information in the user space follows a standard structure.
- List Data is returned in a format that is requested when the API is called.
- All information is accessed by using pointers and offsets, which are identified in the Generic Header, to overlay data structures onto the relevant portions of the user space.

ILE APIs

As the name implies, all of the ILE APIs are bound calls. This means that they can only be called from true ILE programs, for example, programs that are created with the CRTPGM command or by specifying DFTACTGRP(*NO) on the CRTBNDRPG command.

The ILE style APIs come in two formats: regular ILE APIs and the ILE CEE APIs.

Regular ILE APIs

The regular ILE APIs are simply ILE procedure equivalents of some of the regular OPM style API programs. The documentation, definition, parameters, and error processing are the exact same for both. The only difference is that one is a bound call and the other is a dynamic call.

API functions that have an equivalent OPM program and ILE procedure available are identified in the documentation by having two API names listed. Figure 31 on page 53 shows the list of National Language Support-related APIs (APIs by **Category → National Language Support →National Language Support-related APIs**), which show that the Convert Case API has two names: QLGCNVCS and **`QlgConvertCase`**, and the Retrieve Locale Information API has two names: QLGRTVLC and **`QlgRetrieveLocaleInformation`**.

Figure 31 shows the list of *National Language Support-related* APIs.

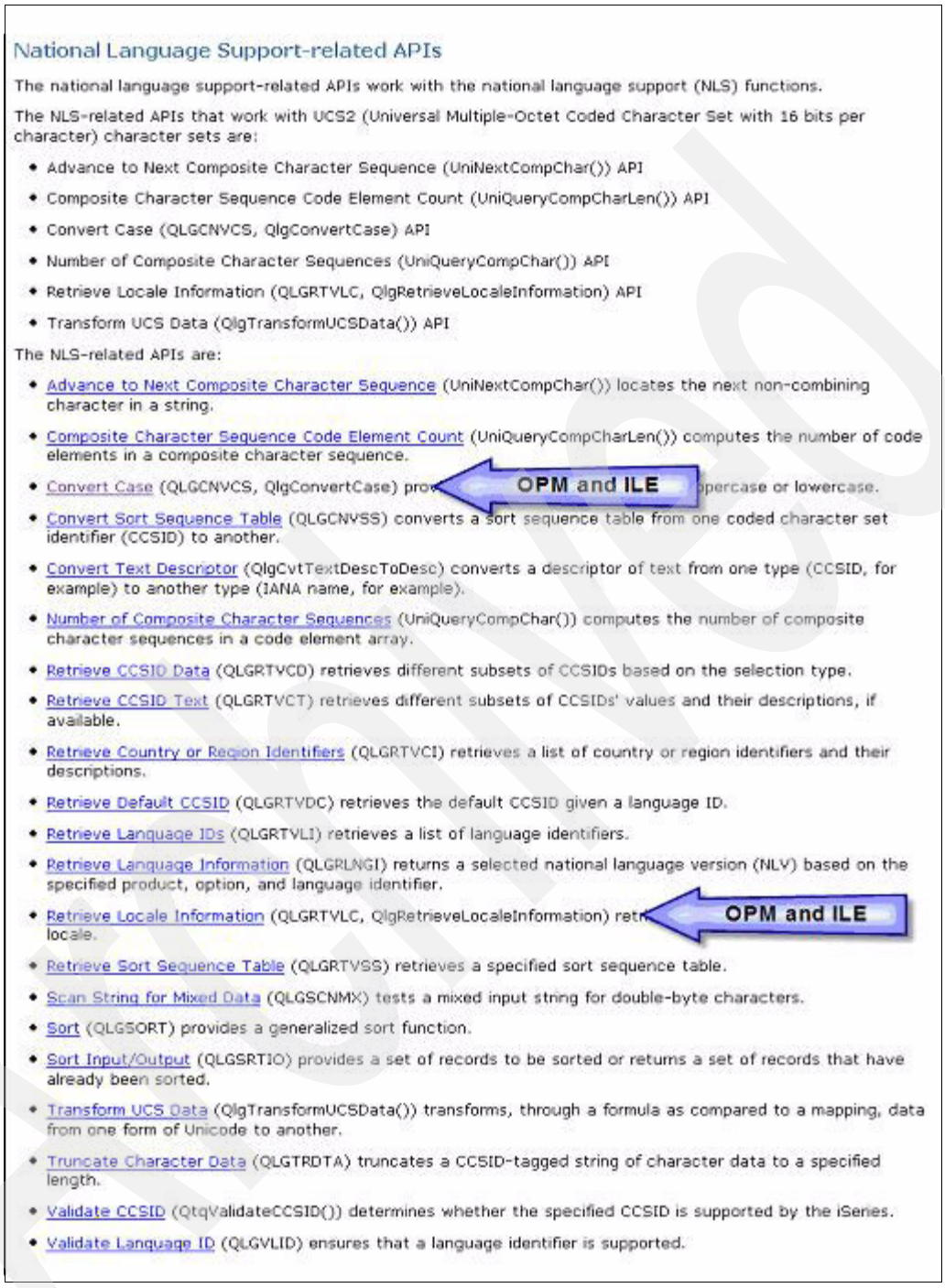


Figure 31 List of National Language Supported Related API

Because we are going to need to convert case in “ILE CEE APIs” on page 57, let us see what is involved in using the **`QlgConvertCase`** API.

Example 35 shows the prototypes for the subprocedures **ConvertToUpper** and **ConvertToLower**, coded in STDAPIINFO, which you can call to convert a string to upper case or lower case. Both APIs accept a string and return a string and allow for an optional second parameter to specify the required CCSID.

Example 35 ConverToUpper and ConvertToLower

```
D ConvertToUpper  PR          65535A   ExtProc('CONVERTTOUPPER')
  D  StringIn                   65535A   Const
  D  ForCCSID                      10I 0 Const
  D                                      Options(*NoPass)

  D ConvertToLower  PR          65535A   ExtProc('CONVERTTOLOWER')
  D  StringIn                   65535A   Const
  D  ForCCSID                      10I 0 Const
  D                                      Options(*NoPass)
```

Figure 32 shows the definition of the *Convert Case* API itself. Again, both the program name and procedure name are identified for the API. The benefit here is that the definition of the parameters is exactly the same for both.

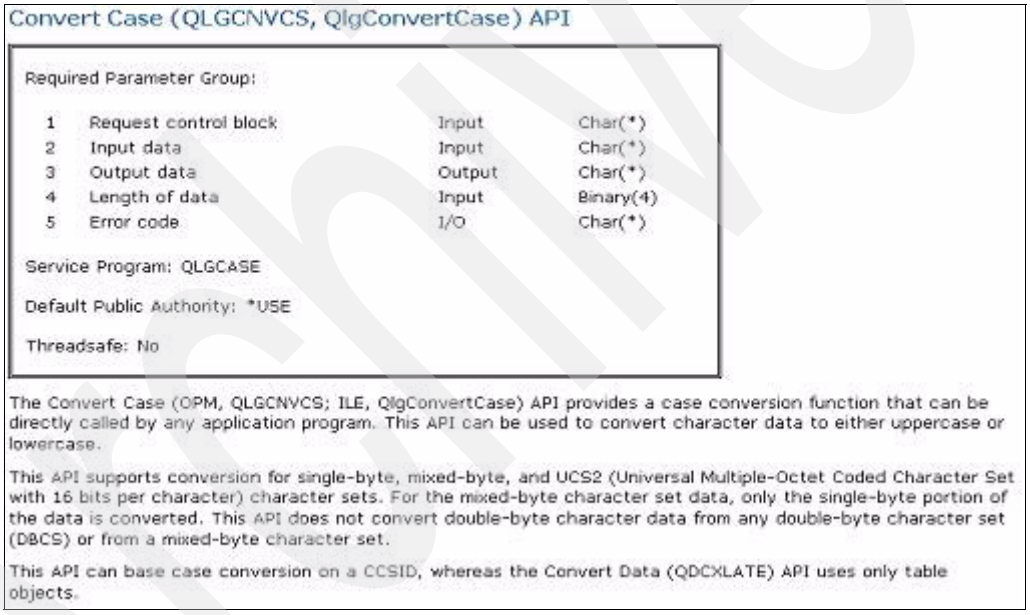


Figure 32 Definition of the Convert Case API with two call names

The service program QLGCASE (identified directly after the parameters) is included automatically when a program is created, but there are some service programs that are not automatically included that you need to identify when you create programs. You know you have a problem if you receive a “CPD5D02 Definition not found for symbol 'name_of_API'” message in the job log when you try to create the program. In such an instance, the easiest way to ensure that the service program is included in the creation process is to include the service program in a binding directory that is specified in the Binding Directory parameter on the create command.

So why would you use one API as opposed to the other? In most cases, it is doubtful that there is much difference between the APIs. There is a chance that the ILE procedure is a more recent edition of the API and might therefore be better written and more efficient, then again, maybe not. There is also the possibility of a performance gain by executing bound calls as opposed to dynamic calls; however, the difference would only really be noticeable if the

API were being called continuously in a loop. The main difference is a simple one of self documentation: It is a lot easier to figure out what the called API does from the name **QlgConvertCase** as opposed to the name QLGCNVCS.

The names of ILE procedures are case sensitive so be sure that you use the EXTPROC keyword to identify the name of the procedure on the prototype, even if you are going to use the same name for the prototype. In RPG IV, if you do not provide an EXTPROC (or EXTPGM) keyword on a prototype, the compiler assumes that the name of the called procedure is the upper case equivalent of the prototype name. The easiest way to ensure you get it right is to copy and paste the name of the procedure from the Web page.

Figure 33 shows the definition of the Request Control Block for the Convert Case API. The layout of the structure is different depending on the value of the first parameter. Always make sure you read the description for every field, for example, you might be inclined to ignore the 10 character reserved value at the end of the layout for the CCSID format, but the documentation specifies that you must initialize these 10 characters to hexadecimal zeros.

Format of Request Control Block

The following table shows the layout of the request control block. For a detailed description of each field, see Field Descriptions.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Type of request
Note: The rest of the layout when the request is 1 (CCSID format)			
4	4	BINARY(4)	CCSID of input data
8	8	BINARY(4)	Case request
12	C	CHAR(10)	Reserved
Note: The rest of the layout when the request is 2 (table object format)			
4	4	BINARY(4)	DBCS indicator
8	8	CHAR(20)	Qualified table name
Note: The rest of the layout when the request is 3 (user-defined format)			
4	4	BINARY(4)	DBCS indicator
8	8	BINARY(4)	Reserved
12	C	BINARY(4)	Length of user-defined table
16	10	CHAR(*)	User-defined case conversion table

Figure 33 Definition of the Request Control Block for the Convert Case API

CASEPROCS is a module that contains two subprocedures (**ConvertToUpper** and **ConvertToLower**) that make use of the **QlgConvertCase** API, as shown in Example 36.

Example 36 CASEPROCS

```
(1)  H NoMain Option(*SrcStmt : *NoDebugIO)

      /Copy APISRC,STDAPIINFO

        // Request 1 - CCSID Format
(2)  D RCB1            DS                  Qualified
     D  RequestType                 10I 0
     D  CCSID                       10I 0
     D  CaseRequest                 10I 0
(3)  D                              10A   Inz(*AllX'00')

(4)  D ConvertCase     PR                  ExtProc('QlgConvertCase')
     D  RCB                             Like(RCB1)
     D                                  Const
     D  InputData                65535A   Const
     D  OutputData               65535A
```

```
     D  DataLength                   10I 0 Const
     D  ErrorCode                          LikeDS(APIError)

        // Convert a string to Upper Case
     P ConvertToUpper  B                   Export
(5)  D ConvertToUpper  PI         65535A
     D  StringIn                  65535A   Const
     D  ForCCSID                     10I 0 Const
     D                                     Options(*NoPass)

     D  StringOut      S          65535A

      /Free
(6)       RCB1.RequestType = 1;
(7)       If (%Parms() > 1);
             RCB1.CCSID = ForCCSID;
          Else;
             RCB1.CCSID = 0;
          EndIf;
(8)       RCB1.CaseRequest = 0;
(9)       ConvertCase( RCB1
                     : StringIn
                     : StringOut
                     : %Len(%Trim(StringIn))
                     : APIError);
(10)      Return StringOut;
      /End-Free
     P                 E

        // Convert a string to Lower Case
     P ConvertToLower  B                   Export
     D ConvertToLower  PI         65535A
     D  StringIn                  65535A   Const
     D  ForCCSID                     10I 0 Const
     D                                     Options(*NoPass)

     D  StringOut      S          65535A

      /Free
          RCB1.RequestType = 1;
          If (%Parms() > 1);
             RCB1.CCSID = ForCCSID;
          Else;
             RCB1.CCSID = 0;
          EndIf;
(11)      RCB1.CaseRequest = 1;
          ConvertCase( RCB1
                     : StringIn
                     : StringOut
                     : %Len(%Trim(StringIn))
                     : APIError);
          Return StringOut;
      /End-Free
     P                 E
```

Create the CASEPROCS service program by using these commands:

```
CRTRPGMOD MODULE(REDBOOK/CASEPROCS) SRCFILE(REDBOOK/APISRC)
CRTSRVPGM SRVPGM(REDBOOK/CASEPROCS) EXPORT(*ALL)
```

Add the service program to the APIS binding directory APIS by using this command:

```
ADDBNDDIRE BNDDIR(REDBOOK/APIS) OBJ((REDBOOK/CASEPROCS))
```

The following list points out the main points in Example 36 on page 55. The numbers in this list correspond to the numbers in Example 36 on page 55:

1. CASEPROCS is a NOMAIN module.
2. RCB1 is the Request Control Block for a CCSID format request. The Request Control Block is defined in CASEPROCS as opposed to STDAPIINFO because we are encapsulating calls to QlgConvertCase through the ConvertToUpper and ConvertToLower subprocedures.
3. The reserved portion of the Request Control Block is initialized to hex '00'.
4. **ConvertCase** is the prototype for **QlgConvertCase**. Note that the use of the EXTPROC keyword as opposed to EXTPGM and the procedure name in mixed case. As with the Request Control Block, the prototype is defined in CASEPROCS as opposed to STDAPIINFO because we are encapsulating calls to **QlgConvertCase** through the **ConvertToUpper** and **ConvertToLower** subprocedures.
5. ConvertToUpper accepts and returns a string. It also allows for an optional CCSID as a second parameter.
6. The Request Type in the Request Control Block is set to 1 for a CCSID format request.
7. The CCSID in the Request Control Block is set to the requested CCSID or defaults to zero if a CCSID was not requested.
8. The Case Request in the Request Control Block is set to zero to request a conversion to upper case.
9. The **QlgConvertCase** API is called.
10. The converted string is returned.
11. **ConvertToLower** is the exact same as **ConvertToUpper** except that the Case Request in the Request Control Block is set to one to request a conversion to lower case.

We show an example of using **ConvertToUpper** in the next section.

ILE CEE APIs

The *ILE CEE APIs* are easy to identify because their names all start with either “CEE” or “CE4”. The APIs that start with “CEE” might have a corresponding API on another IBM platform; whereas, those APIs that start with “CE4” are specific to OS/400 or i5/OS.

The documentation for the ILE CEE APIs, the definition of parameters, and the way in which errors are handled are different from the traditional APIs. The good news is that the ILE CEE APIs are very well documented.

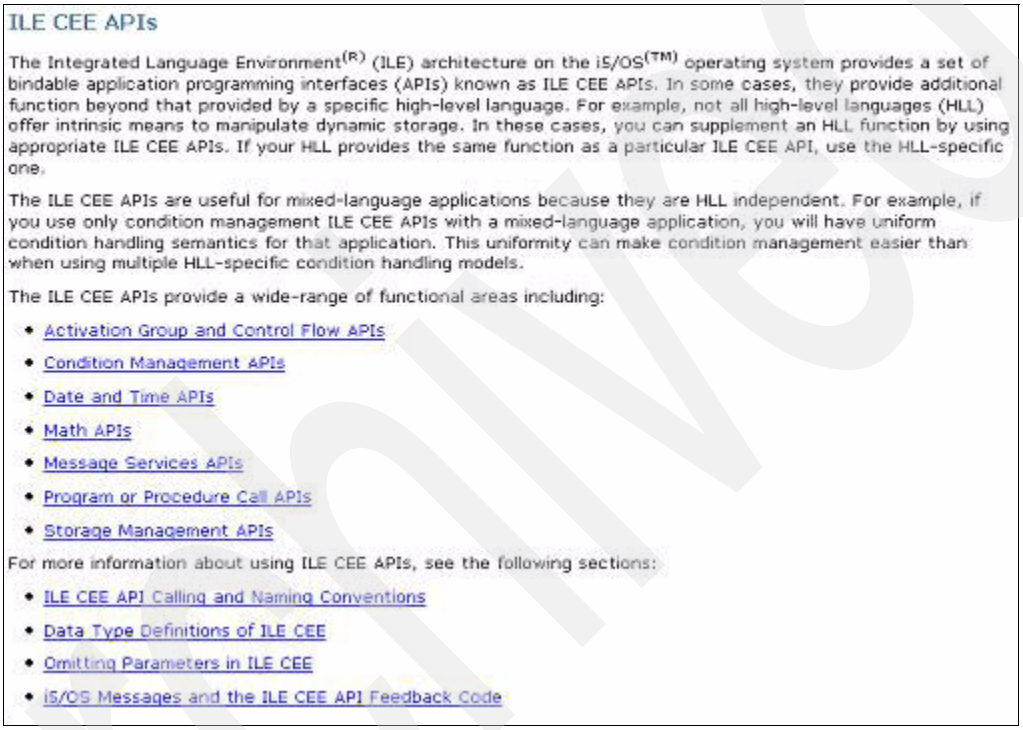
The first step is to understand how the data types for the ILE CEE APIs are identified in the documentation and how to represent them in RPG IV:

1. From the Information Center, access the Application programming interfaces page at the following Web site:

   https://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/apiref/api.htm

2. Click the **APIs by Category** link, and then click the **ILE CEE** link. The ILE CEE APIs page is displayed (Figure 34).

3. Click the **Data Type Definitions for ILE CEE** link, and the page shown in Figure 34 is displayed.



ILE CEE APIs

The Integrated Language Environment(R) (ILE) architecture on the i5/OS(TM) operating system provides a set of bindable application programming interfaces (APIs) known as ILE CEE APIs. In some cases, they provide additional function beyond that provided by a specific high-level language. For example, not all high-level languages (HLL) offer intrinsic means to manipulate dynamic storage. In these cases, you can supplement an HLL function by using appropriate ILE CEE APIs. If your HLL provides the same function as a particular ILE CEE API, use the HLL-specific one.

The ILE CEE APIs are useful for mixed-language applications because they are HLL independent. For example, if you use only condition management ILE CEE APIs with a mixed-language application, you will have uniform condition handling semantics for that application. This uniformity can make condition management easier than when using multiple HLL-specific condition handling models.

The ILE CEE APIs provide a wide-range of functional areas including:

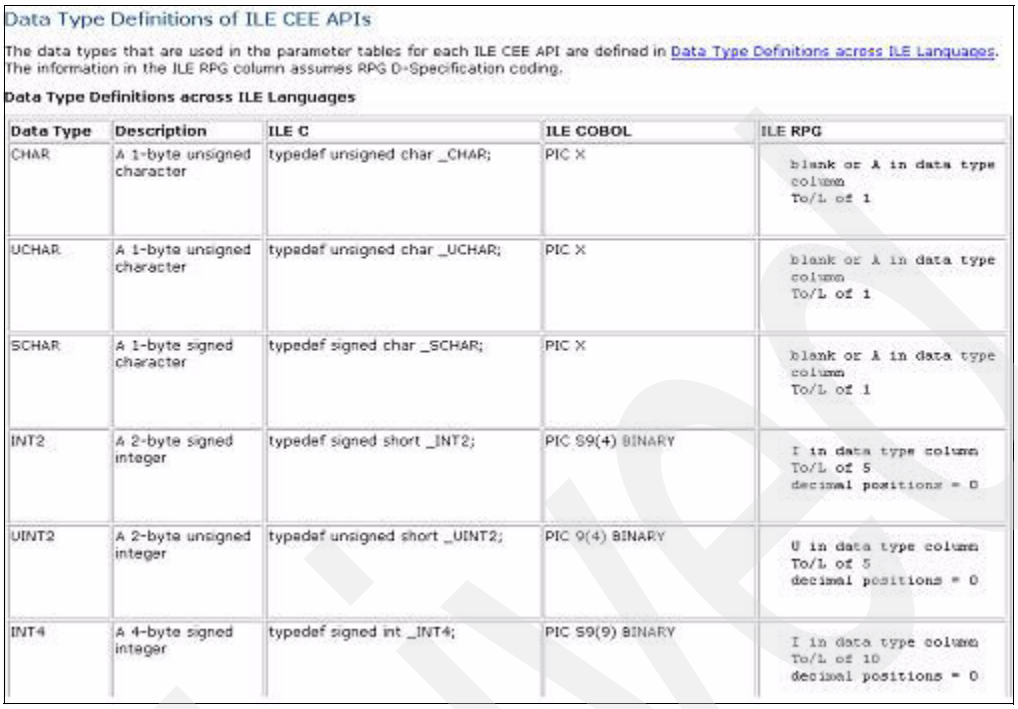
- Activation Group and Control Flow APIs
- Condition Management APIs
- Date and Time APIs
- Math APIs
- Message Services APIs
- Program or Procedure Call APIs
- Storage Management APIs

For more information about using ILE CEE APIs, see the following sections:

- ILE CEE API Calling and Naming Conventions
- Data Type Definitions of ILE CEE
- Omitting Parameters in ILE CEE
- i5/OS Messages and the ILE CEE API Feedback Code

Figure 34 ILE CEE APIs

Figure 35 shows the start of the Data Type Definitions of ILE CEE APIs page. The table shows how each data type is referenced in ILE C, ILE COBOL, and ILE RPG.



Data Type Definitions of ILE CEE APIs

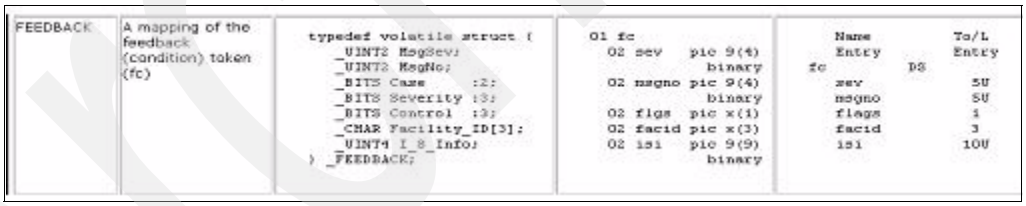
The data types that are used in the parameter tables for each ILE CEE API are defined in Data Type Definitions across ILE Languages. The information in the ILE RPG column assumes RPG D-Specification coding.

Data Type Definitions across ILE Languages

Data Type	Description	ILE C	ILE COBOL	ILE RPG
CHAR	A 1-byte unsigned character	typedef unsigned char _CHAR;	PIC X	blank or A in data type column To/L of 1
UCHAR	A 1-byte unsigned character	typedef unsigned char _UCHAR;	PIC X	blank or A in data type column To/L of 1
SCHAR	A 1-byte signed character	typedef signed char _SCHAR;	PIC X	blank or A in data type column To/L of 1
INT2	A 2-byte signed integer	typedef signed short _INT2;	PIC S9(4) BINARY	I in data type column To/L of 5 decimal positions = 0
UINT2	A 2-byte unsigned integer	typedef unsigned short _UINT2;	PIC 9(4) BINARY	U in data type column To/L of 5 decimal positions = 0
INT4	A 4-byte signed integer	typedef signed int _INT4;	PIC S9(9) BINARY	I in data type column To/L of 10 decimal positions = 0

Figure 35 Start of the Data Type Definitions of ILE CEE APIs

Further down the Data Type Definitions of the ILE CEE APIs page is the definition of the feedback token, as shown in Figure 36. The feedback token is the ILE CEE API equivalent of the API error structure for standard APIs. It provides feedback information for any error conditions that are encountered by the called API.



FEEDBACK	A mapping of the feedback (condition) token (fc)	typedef volatile struct { _UINT2 MsgSev; _UINT2 MsgNo; _BITS Case :2; _BITS Severity :3; _BITS Control :3; _CHAR Facility_ID[3]; _UINT4 I_S_Info; } _FEEDBACK;	01 fc 02 sev pic 9(4) binary 02 msgno pic 9(4) binary 02 flgs pic x(1) 02 facid pic x(3) 02 isi pic 9(9) binary	Name Entry To/L Entry fc DS sev 5U msgno 5U flags 1 facid 3 isi 10U

Figure 36 Definitions of the feedback token

STDAPIINFO contains the corresponding definition of the feedback token, as shown in Example 37.

Example 37 Definition of the feedback token

```
D fc              DS                  Qualified
D  sev                               5U 0
D  msgno                             5U 0
D   msgnochar                        2A   Overlay(msgno)
D  flags                             1A
D  facid                             3A
D  isi                              10U 0
```

The subfield facid provides the first three characters of the message ID and the subfield msgno provides the last four. In the documentation, the definition of the message number

(msgno) is slightly misleading. It should be a two-byte hexadecimal number, which unfortunately, cannot be defined in RPG IV. To determine the true value of msgno you must convert it from hex to character. You may be tempted to try this yourself (for example, using the MI Instruction API **cvthc**—Convert Hex to Decimal), but there is another ILE CEE API that makes the deciphering of the feedback token very straightforward:

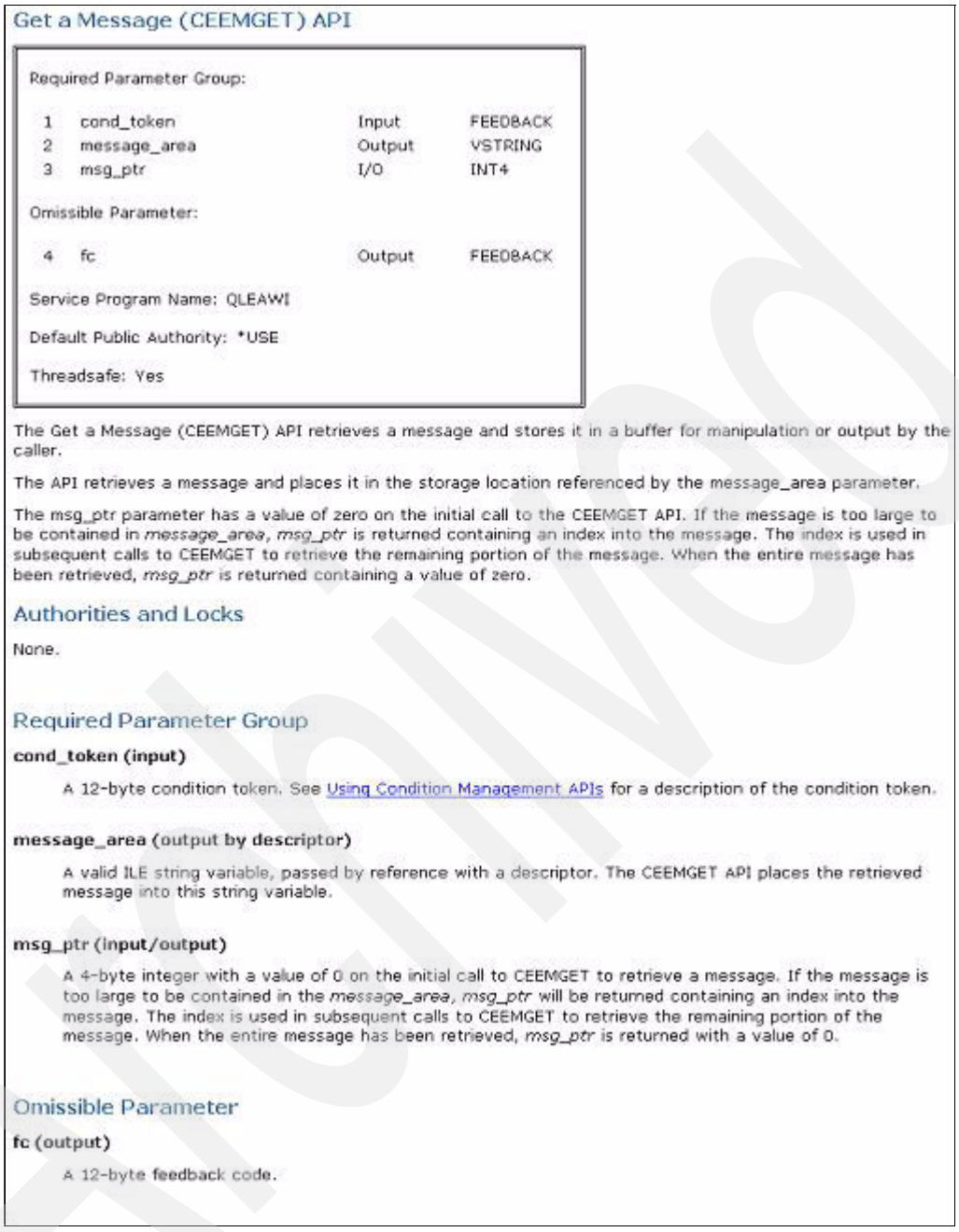
1. From the ILE CEE APIs page, click the **Message Services APIs** link.

   https://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/apiref/api.htm

2. Click the **Get a Message (CEEMGET)** link, and Figure 37 on page 61 is displayed.
3. You pass the API a feedback token and it returns the text of the message. The API may be called multiple times if the amount of text that can be returned exceeds the size of the field you supply as the second parameter.

Important: Always read the description of the ILE CEE APIs very carefully: Note how the second parameter is “passed by reference with descriptor”, which means that you must specify the OPDESC keyword on the prototype for the API.

Figure 37 shows the Get a Message (CEEMGET) API.



Get a Message (CEEMGET) API

Required Parameter Group:

1	cond_token	Input	FEEDBACK
2	message_area	Output	VSTRING
3	msg_ptr	I/O	INT4

Omissible Parameter:

4	fc	Output	FEEDBACK

Service Program Name: QLEAWI

Default Public Authority: *USE

Threadsafe: Yes

The Get a Message (CEEMGET) API retrieves a message and stores it in a buffer for manipulation or output by the caller.

The API retrieves a message and places it in the storage location referenced by the message_area parameter.

The msg_ptr parameter has a value of zero on the initial call to the CEEMGET API. If the message is too large to be contained in *message_area*, *msg_ptr* is returned containing an index into the message. The index is used in subsequent calls to CEEMGET to retrieve the remaining portion of the message. When the entire message has been retrieved, *msg_ptr* is returned containing a value of zero.

Authorities and Locks

None.

Required Parameter Group

cond_token (input)

A 12-byte condition token. See Using Condition Management APIs for a description of the condition token.

message_area (output by descriptor)

A valid ILE string variable, passed by reference with a descriptor. The CEEMGET API places the retrieved message into this string variable.

msg_ptr (input/output)

A 4-byte integer with a value of 0 on the initial call to CEEMGET to retrieve a message. If the message is too large to be contained in the *message_area*, *msg_ptr* will be returned containing an index into the message. The index is used in subsequent calls to CEEMGET to retrieve the remaining portion of the message. When the entire message has been retrieved, *msg_ptr* is returned with a value of 0.

Omissible Parameter

fc (output)

A 12-byte feedback code.

Figure 37 Get a Message (CEEMGET) API

STDAPIINFO contains the corresponding prototype definition, as shown in Example 38.

Example 38 Prototype definition

```
D GetAMessage     PR                  ExtProc('CEEMGET')
(1)  D                                     OpDesc
(2)  D  cond_token                         LikeDS(fc)
     D                                     Const
(3)  D  message_area               50A     Varying
     D  msg_ptr                    10I 0
(4)  D  fcOut                              LikeDs(fc)
     D                                     Options(*Omit)
```

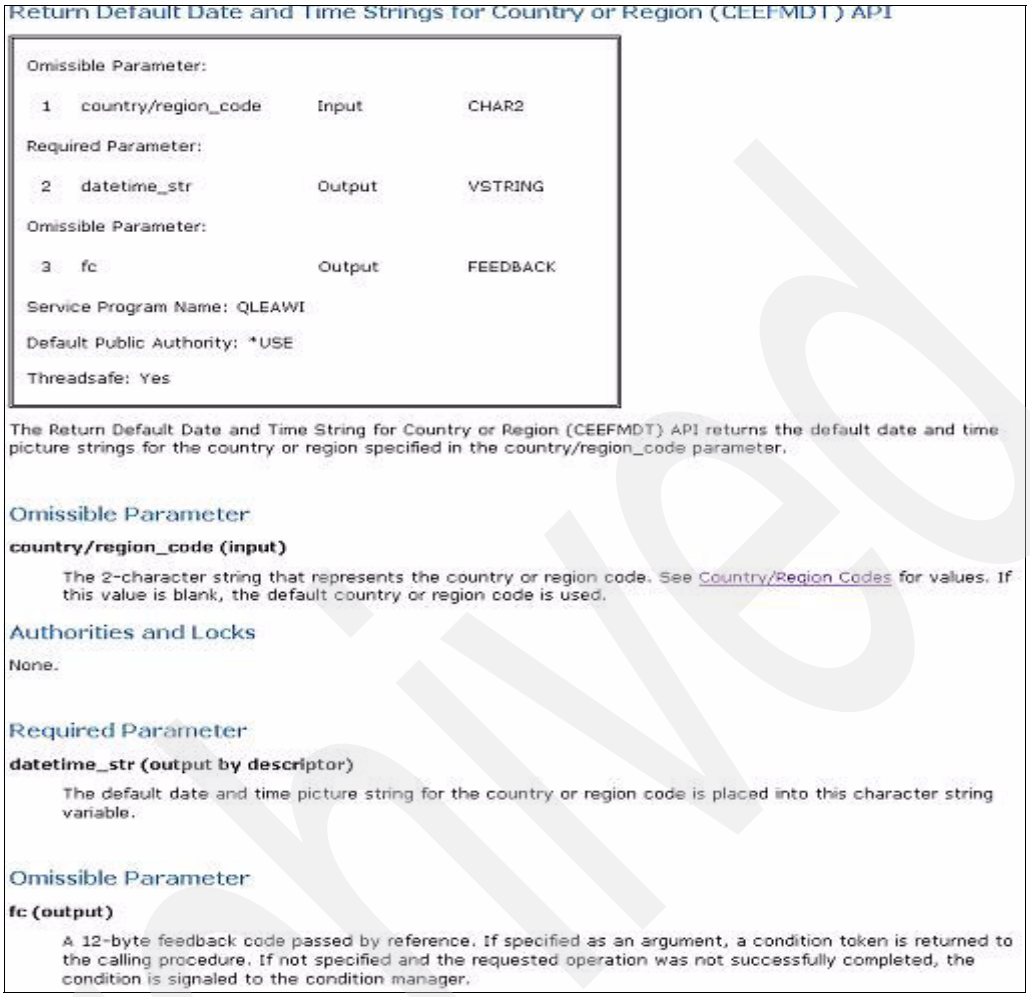
The following list contains the main points to note in Example 38. The numbers in this list correspond with the numbers in Example 38:

1. The OPDESC keyword is required because at least one parameter requires an operational descriptor.
2. The condition token is a feedback token.
3. The message text returned is only defined as 50 characters because we are using the DSPLY operation to display its contents (DSPLY may only display up to 52 characters of text).
4. The API might, itself, return a feedback token. You can omit the feedback token as opposed to being optional.

Section 9.6 from *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402 provides examples of how to use the ILE CEE APIs for Condition Management, Activation Group, and Control Flow. As an example of deciphering an ILE CEE API:

1. From the ILE CEE APIs page, click the **Date and Time APIs** link.
2. Click the **Return Default Date and Time Strings for Country or Region (CEEFMDT) API** link. Figure 38 on page 63 is displayed. This API returns a date and time picture string for the requested country or region code. A list of valid country and region codes is provided at the bottom of the page for the API.

Figure 38 shows the Return Default and Time Strings for Country or Region API.


Return Default Date and Time Strings for Country or Region (CEEFMDT) API

Omissible Parameter:

1	country/region_code	Input	CHAR2

Required Parameter:

2	datetime_str	Output	VSTRING

Omissible Parameter:

3	fc	Output	FEEDBACK

Service Program Name: QLEAWI

Default Public Authority: *USE

Threadsafe: Yes

The Return Default Date and Time String for Country or Region (CEEFMDT) API returns the default date and time picture strings for the country or region specified in the country/region_code parameter.

Omissible Parameter

country/region_code (input)

The 2-character string that represents the country or region code. See Country/Region Codes for values. If this value is blank, the default country or region code is used.

Authorities and Locks

None.

Required Parameter

datetime_str (output by descriptor)

The default date and time picture string for the country or region code is placed into this character string variable.

Omissible Parameter

fc (output)

A 12-byte feedback code passed by reference. If specified as an argument, a condition token is returned to the calling procedure. If not specified and the requested operation was not successfully completed, the condition is signaled to the condition manager.

Figure 38 Return Default Date and Time Strings for Country or Region (CEEFMDT) API

STDAPIINFO contains the corresponding prototype definition, which we show in Example 39.

Example 39 Prototype definition for CEEFMDT API

```
D DateTimeForCountry...
     D                 PR                  ExtProc('CEEFMDT')
(1)  D                                     OpDesc
(2)  D  CountryCode                  2A    Options(*OMIT)
     D  time_pic_str                 40A   Varying
     D  fcOut                             LikeDs(fc)
     D                                    Options(*Omit)
```

The numbers in the following list correspond with the numbers in Example 39. The main points to note are:

1. The OPDESC keyword is required since at least one parameter requires an operational descriptor.
2. If the country code is blank or omitted the default country code or region code is used.

API04, Example 40, is a program that demonstrates the use of the CEEFMDT and CEEMGET APIs along with the **QlgConvertCase** API that we described in “Regular ILE APIs” on page 52. API04 allows for the entry of a country or region code and displays the corresponding date and time format for the country or region. If there is an error on a call to CEEFMDT, the required message text is retrieved and displayed.

Example 40 AP104

```
(1)  H DftActGrp(*NO) ActGrp(*New) BndDir('APIS')
     H Option(*SrcStmt:*NoDebugIO)

      /Copy APISRC,STDAPIINFO

     D CountryID       S              2A
     D DateTime        S             40A   Varying
     D MsgText         S             50A   Varying
     D MsgPtr          S             10I 0

      /Free
(2)      DoU CountryId = '**';
            CountryId = *Blanks;
(3)         Dsply 'Enter Country Code (** to End)'
                  ' ' CountryId;

            If (CountryId <> '**');
(4)            CountryId = ConvertToUpper(CountryId);
(5)            DateTimeForCountry( CountryID
                                 : DateTime
                                 : fc);
               If fc.sev=0;
(6)               Dsply DateTime;
               Else;
(7)               MsgPtr = 0;
                  DoU MsgPtr = 0;
                     GetAMessage( fc
                                : MsgText
                                : MsgPtr
                                : *OMIT);
                     Dsply MsgText;
                  EndDo;
               EndIf;
            EndIf;
         EndDo;

         *InLR = *On;
      /End-Free
```

Create the program in Example 40 by using the command:

```
CRTBNDRPG PGM(REDBOOK/API04) SRCFILE(REDBOOK/APISRC)
```

The numbers in the following list correspond with the numbers in Example 40 on page 64. The main points to note in Example 40 on page 64 are:

1. The APIS binding directory is specified in the H spec because the program makes use of the **ConvertToUpper** subprocedure in the CASEPROCS service program. You do not need to specify a service program for the ILE CEE APIs because they are automatically bound.
2. The program loops until a value of '**' is entered for the country code.
3. A country or region code is entered.
4. The entered code is converted to uppercase.
5. The CEEFMDT API is called to retrieve the date time format.
6. If the call to CEEFMDT was successful, the date time format is displayed.
7. If the call to CEEFMDT was not successful, the CEEMGET API is called to retrieve the message text for the message that was identified in the feedback token. The message pointer parameter is greater then zero if more message text is available. Its value is the starting position of the next bit of available text. A subsequent call to CEEMGET retrieves the text that starts at that position.

Calling the program API04 and providing country IDs of blank (for default, which happens to be US in this example), for example, (Ireland), us (United States) and jw (invalid) results in the following being displayed.

```
DSPLY  Enter Country Code (** to End)
*N
DSPLY  MM/DD/YY ZH:MI:SS AP
DSPLY  Enter Country Code (** to End)
ie
DSPLY  DD/MM/YY ZH:MI:SS
DSPLY  Enter Country Code (** to End)
us
DSPLY  MM/DD/YY ZH:MI:SS AP
DSPLY  Enter Country Code (** to End)
jw
DSPLY  The country/region_code identifier is not valid fo
DSPLY  r CEEFMDT.
DSPLY  Enter Country Code (** to End)
```

Note how the message text for the invalid country ID is split over two lines.

Although they read a little differently from the traditional APIs, the ILE CEE APIs are not that difficult to interpret.

UNIX-Type APIs

The *UNIX-Type APIs* are so called because they are based on standard APIs that are available on UNIX systems. The UNIX-Type APIs have a different style of documentation and error handling just as the ILE CEE APIs do. Unfortunately for the RPG programmer, the documentation for the UNIX-Type APIs is not as easy to comprehend as the documentation for the ILE CEE APIs.

The UNIX-Type APIs are documented with the C/C++ programmer in mind, which makes some sense when you consider that the APIs were ported from UNIX where C/C++ is the norm; however, RPG and COBOL are still the norm on System i™, and unless you have

knowledge of C/C++, you are going to find the documentation a challenge—but not an insurmountable challenge.

Before tackling the APIs, familiarize yourself with simply calling C functions. The best place to start is in Section 5.1 “Exploiting the C function library: a case study” in *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402.

It is easy to interpret some of the basic data types used as parameters, which are described in section 5.1.1 of *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402; however, you may come across data types that do not seem to make any sense. Help is at hand. Barbara Morris of the IBM Toronto labs has a Web page titled “Converting from C prototypes to RPG prototypes,” which gives you most of what you need. But do not jump straight to this Web page. It is best to familiarize yourself with a few of the simpler APIs before tackling parameter definitions that require Barbara's clarification.

You can find the Web page at:

`http://www.opensource400.org/callc.html`

We also advise that you install the System Includes Library (QSYSINC) on your system. You will almost certainly want to refer to some of the system supplied sources. You can install QSYSINC on your system by installing Option 13 of the Operating System (5722SS1) - System Openness Includes.

Interpreting the interface

Let us look at an example of using a UNIX-Type API:

1. From the Information Center (Figure 34 on page 58), access the Application programming interfaces page at the following Web site:

   `https://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/apiref/api.htm`

2. Click the **APIs by Category** link, and then click the **UNIX-Type** link. The UNIX-Type APIs page is displayed.
3. From the UNIX-Type APIs page, click the **Integrated File System APIs** link, and then click the access() **(Determine file accessibility)** link, which is the very first API listed. The window in Figure 39 is displayed. Use the access() API to determine if a path or file exists and whether or not you can access it, rather like using the Check Object (CHKOBJ) command for the IFS.

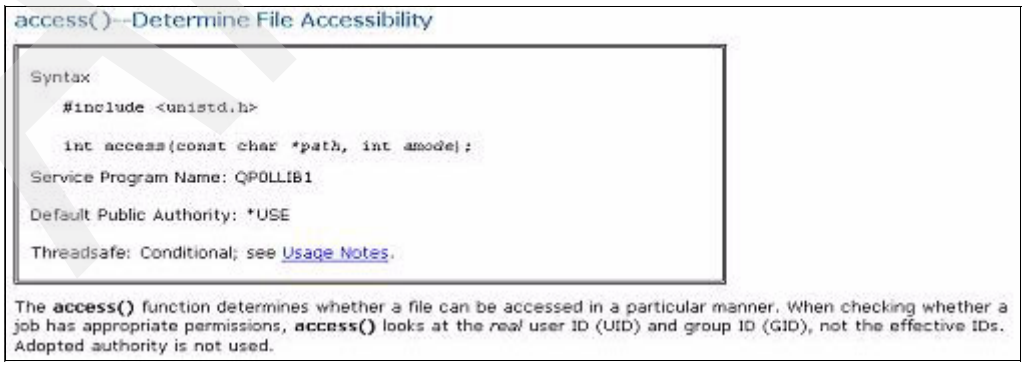
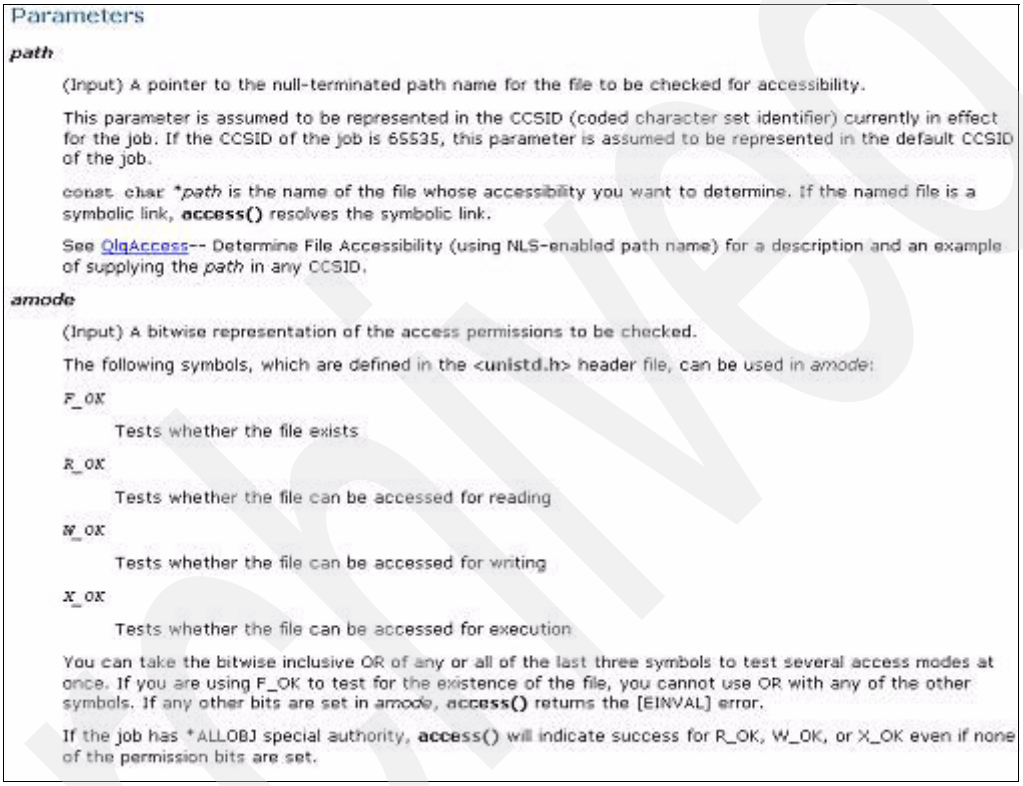


Figure 39 Syntax for the access() API

The access() Determine File Accessibility is not quite the same as the documentation for the previous APIs! The format that is used to describe the interface for the API is actually C code, which is all well and good if you are a C programmer.

The second line, which reads **`int access(const char *path, int amode)`**, describes the interface for the API. It indicates that access() returns an integer (int at the start of the line) and accepts two parameters: a pointer to a null terminated path name (const char *path) and an integer that indicates the required access mode (int amode).

The next part of the documentation provides a description of the parameters, as shown in Figure 40.



Parameters

path

(Input) A pointer to the null-terminated path name for the file to be checked for accessibility.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

const char *path is the name of the file whose accessibility you want to determine. If the named file is a symbolic link, **access()** resolves the symbolic link.

See QlgAccess-- Determine File Accessibility (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

amode

(Input) A bitwise representation of the access permissions to be checked.

The following symbols, which are defined in the <unistd.h> header file, can be used in *amode*:

F_OK

Tests whether the file exists

R_OK

Tests whether the file can be accessed for reading

W_OK

Tests whether the file can be accessed for writing

X_OK

Tests whether the file can be accessed for execution

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access modes at once. If you are using F_OK to test for the existence of the file, you cannot use OR with any of the other symbols. If any other bits are set in *amode*, **access()** returns the [EINVAL] error.

If the job has *ALLOBJ special authority, **access()** will indicate success for R_OK, W_OK, or X_OK even if none of the permission bits are set.

Figure 40 Parameters for the access() API

The definition of the first parameter specifies “a pointer to the null-terminated path name.” In RPG, we are used to fixed-length strings (for example, we predefine the maximum length of character fields), but in C, the length of a string is indeterminate, and the length is based on what the content of the string is. C strings are terminated with a null value to indicate the end of the string, which you must take into account when you call C functions from RPG. Use OPTIONS(*STRING) on the definition of the parameter in the prototype, and use the %STR built in function to retrieve a string returned from a C function.

The definition of the second parameter reads a lot more complicated then it actually is. A lot of the APIs written in C use a method of setting bits in a byte to indicate certain requirements. The different permutation and combination of bits are defined as named constants. The documentation tells you that the values you can specify for the second parameter are represented by one of the named constants (F_OK, R_OK, W_OK or X_OK), which are defined in the member unistd.h.

If you were writing a C program, you could simply include the unistd.h member in your program (just like using a copy member in RPG); however, because you are writing an RPG

program, you have a little work to do. You need to browse the member UNISTD in the source physical file H in the library QSYSINC, and search for “constants for access” to see the following definitions.

```
/*****************************************************************/
/*   Constants for access()                                      */
/*****************************************************************/

#define R_OK  4                   /* Test for read permission     */
#define W_OK  2                   /* Test for write permission    */
#define X_OK  1                   /* Test for execute or search
                                     permission                   */
#define F_OK  0                   /* Test for existence of a file */
```

You now need to redefine these in RPG (more in a moment).

Further down the documentation page, (skipping over the Authorities for the moment) is a description of the return value, as shown in Figure 41.

Figure 41 Return Value for the access() API

A return value of zero means that the call to access() was successful, and a returned value of “-1” means it was not successful. We discuss more about identifying errors in the next section.

You now have enough information to define a prototype and some named constants.

STDAPIINFO contains the definition of the following prototype and named constant definitions, as shown in Example 41.

Example 41 Prototype and named constant definitions

```
(1)  D access          PR            10I 0 ExtProc('access')
(2)  D  path                           *   Value Options(*String)
     D  amode                        10I 0 Value

(3)  D R_OK            C                   4
     D W_OK            C                   2
     D X_OK            C                   1
     D F_OK            C                   0
```

The numbers in the following list correspond with the section numbers in Example 41. The main points to note in Example 41are:

1. Being written in C, the API names are case sensitive so ensure that you use the EXTPROC keyword to specify the API name.
2. The OPTIONS(*STRING) keyword ensures that a null terminator is appended to the parameter value.
3. Named constants are defined to correspond to the definition in UNISTD.

Identifying errors

Figure 41 on page 68 indicates the means by which errors are identified. The API returns a value (usually a negative number) to indicate that the call detected an error. The documentation indicates that the errno global variable is set to indicate the error, which means that you must use the C function **__errno** (that is two underlines followed by “errno”) to retrieve the value of the global variable errno. STDAPIINFO contains the following definitions:

```
D Geterrno        PR              *   ExtProc('__errno')

D p_errno         S               *
D errno           S             10I 0 Based(p_errno)
```

Geterrno returns a pointer to a variable. The value of an integer based on the returned pointer will be the value of the global errno variable. Using **__errno** requires the inclusion of the QC2LE binding directory.

But what values can **errno** contain? The documentation for the API lists the possible error conditions that may be indicated in **errno**, as shown in Figure 42.

Figure 42 shows the error conditions for the access() API.

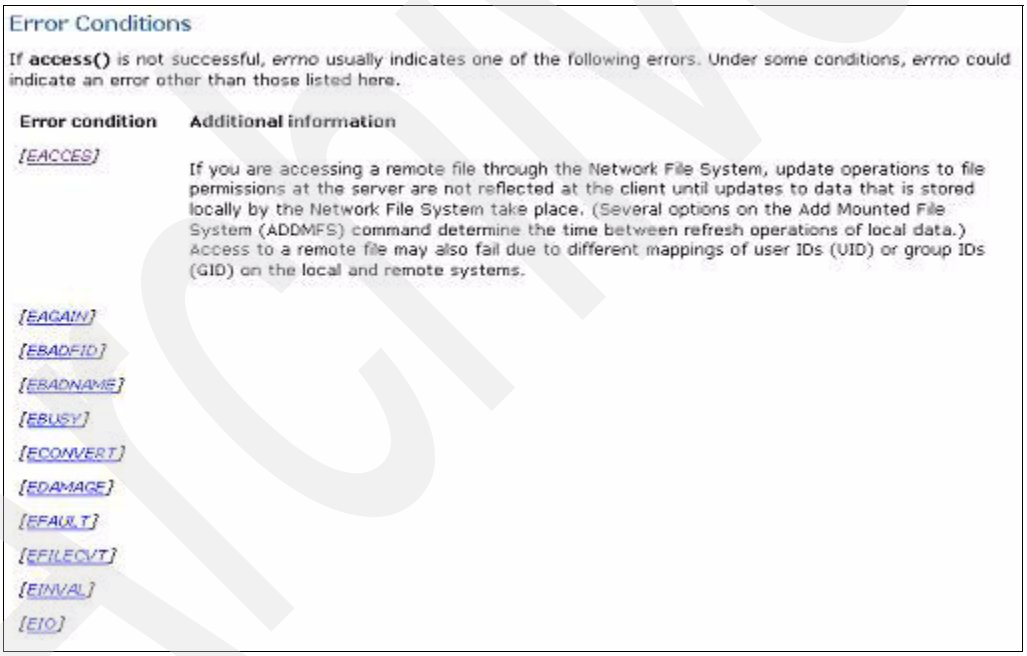
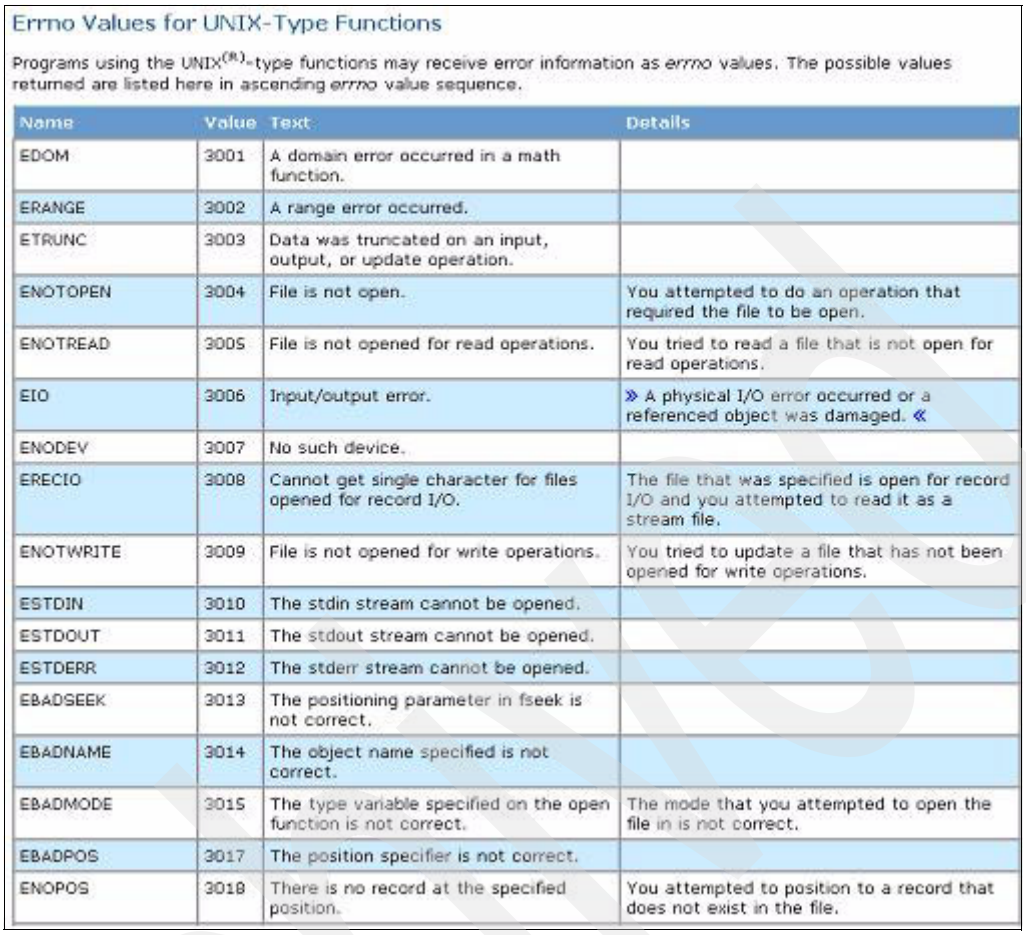


Figure 42 Some of the error conditions for the access() API

Again, C identifies the possible values as named constants. The member ERROR, in the source physical file H in the library QSYSINC, contains the definition of the error condition constants, so you may want to consider converting the member ERROR to RPG. If that seems like too much work, visit Scott Klements Web site, and download the IFSIO_H and ERRNO_H source members from Scott’s RPG and IFS presentation. You can find Scott’s Web site at:

http://www.scottklement.com/presentations/

Alternatively, you can simply select an error condition in the documentation to be shown a table of all the possible Errno Values for UNIX-Type Functions, as shown in Figure 43.



Errno Values for UNIX-Type Functions

Programs using the UNIX(R)-type functions may receive error information as *errno* values. The possible values returned are listed here in ascending *errno* value sequence.

Name	Value	Text	Details
EDOM	3001	A domain error occurred in a math function.	
ERANGE	3002	A range error occurred.	
ETRUNC	3003	Data was truncated on an input, output, or update operation.	
ENOTOPEN	3004	File is not open.	You attempted to do an operation that required the file to be open.
ENOTREAD	3005	File is not opened for read operations.	You tried to read a file that is not open for read operations.
EIO	3006	Input/output error.	» A physical I/O error occurred or a referenced object was damaged. «
ENODEV	3007	No such device.	
ERECIO	3008	Cannot get single character for files opened for record I/O.	The file that was specified is open for record I/O and you attempted to read it as a stream file.
ENOTWRITE	3009	File is not opened for write operations.	You tried to update a file that has not been opened for write operations.
ESTDIN	3010	The stdin stream cannot be opened.	
ESTDOUT	3011	The stdout stream cannot be opened.	
ESTDERR	3012	The stderr stream cannot be opened.	
EBADSEEK	3013	The positioning parameter in fseek is not correct.	
EBADNAME	3014	The object name specified is not correct.	
EBADMODE	3015	The type variable specified on the open function is not correct.	The mode that you attempted to open the file in is not correct.
EBADPOS	3017	The position specifier is not correct.	
ENOPOS	3018	There is no record at the specified position.	You attempted to position to a record that does not exist in the file.

Figure 43 Errno Values for UNIX-Type Functions

For the upcoming example, STDAPIINFO contains the following definition:

```
D ENOENT          C                   3025
D EACCES          C                   3401
```

ENOENT indicates “No Path or Entry” and EACCES indicates “Permission denied.”

Other Documentation

Before we look at an example, let us look at the rest of the documentation that is shown for the access() API.

Figure 44 shows the Authorities that are required for each type of access mode request.

Authorities

Authorization Required for access()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be tested	*X	EACCES
Object when R_OK is specified	*R	EACCES
Object when W_OK is specified	*W	EACCES
Object when X_OK is specified	*X	EACCES
Object when R_OK | W_OK is specified	*RW	EACCES
Object when R_OK | X_OK is specified	*RX	EACCES
Object when W_OK | X_OK is specified	*WX	EACCES
Object when R_OK | W_OK | X_OK is specified	*RWX	EACCES
Object when F_OK is specified	None	None

Figure 44 Authorities for the access() API

Figure 45 shows the error messages that may be sent from the function.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Figure 45 Error Messages for the access() API

Figure 46 shows the Usage Notes for the function. Usage notes highlight any special conditions or considerations for using the function.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400

2. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

3. QOPT File System Differences

   If the object exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object and preceding directories in the path name follows the rules described in Authorization Required for access() . If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or preceding directories. The volume authorization list is checked for the requested authority regardless of the volume media format.

Figure 46 Usage Notes for the access() API

Figure 47 shows related information for the function. Related information lists include files and other functions that relate to the functionality of this function.

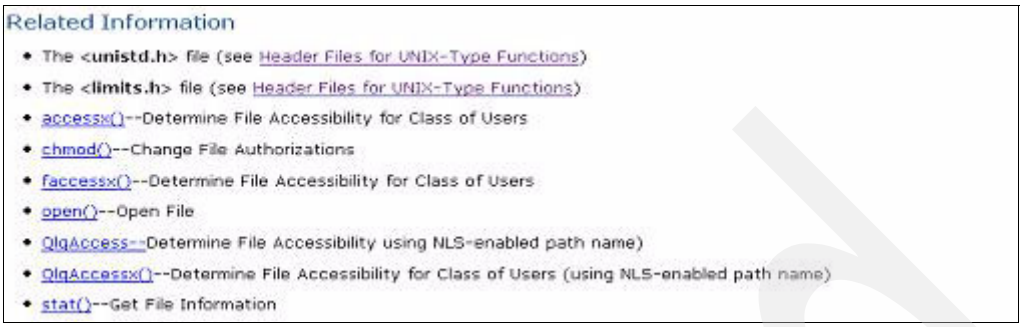


Figure 47 Related information for the access() API

Figure 48 shows an example for the function. Although the example is written in C, you do not need to be an expert in C to get an idea of how you must call the function.

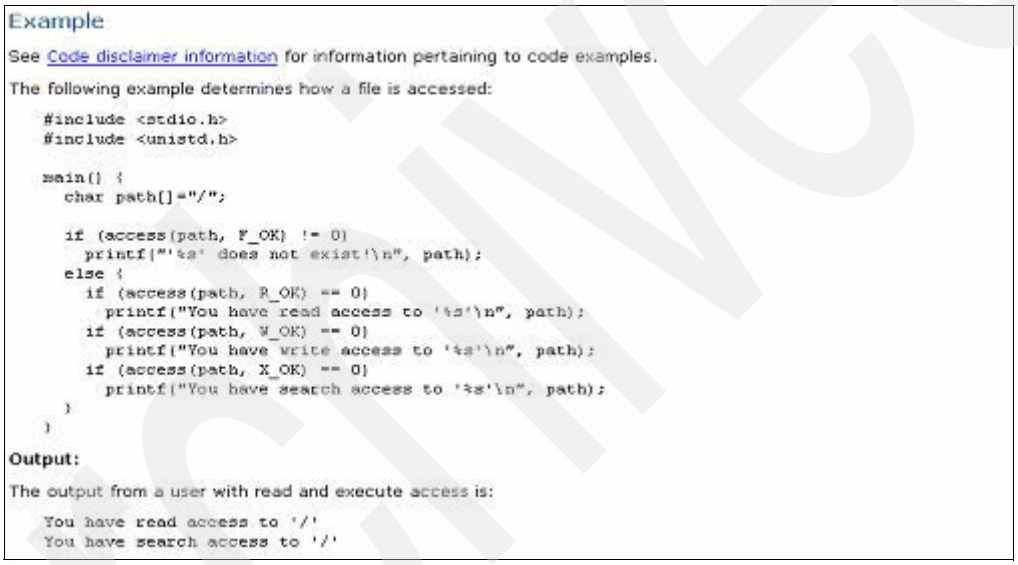


Figure 48 Example for the access() API

Next, we look at an example of using the access() API.

Example of using a UNIX-Type API

API05 is a program that allows for the entry of a path name and indicates whether or not the path exists, as shown in Figure 49.

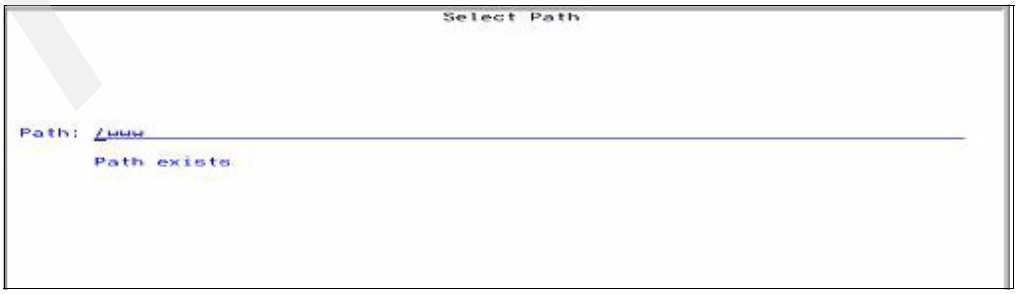


Figure 49 Using the access() API to check the existence of objects in the IFS

API05DSPF contains the definition of the display file, as shown in Example 42.

Example 42 Definition of the display file as contained in AP105DSPF

```
     A* API05D - Display File for access() API Example

    A                                      DSPSIZ(24 80 *DS3)
    A                                      INDARA

    A          R SELECTPATH
    A                                      CA03(03 'Exit')
    A                                    1 36'Select Path'
    A                                      DSPATR(HI)
    A                                    9  2'Path:'
    A            PATHNAME      70   B  9  8CHECK(LC)
    A            MESSAGE       50   O 11  8
    A                                   23  2'F3=Exit'
    A                                      DSPATR(HI)
    A                                      COLOR(BLU)
```

Create this display file by using the command:

```
CRTDSPFF FILE(REDBOOK/API05DSPF) SRCFILE(REDBOOK/APISRC)
```

API05 contains the definition of the program, as shown in Example 43.

Example 43 Definition of the program contained in AP105

```
(1)  H DftActGrp(*NO) ActGrp(*New) BndDir('QC2LE')
     H Option(*SrcStmt:*NoDebugIO)
(2)  FAPI05DSPF CF   E            WorkStn IndDs(WorkStnInd)
(3)   /Copy APISRC,STDAPIINFO
(4)   /Copy APISRC,WORKSTNIND

      /Free
          ExFmt SelectPath;
(5)       DoW Not F3Exit;
(6)          If access(%Trim(PathName):F_OK) = 0;
                 Message = 'Path exists';
             Else;
(7)              p_errno = Geterrno();
                 Select;
                 When (errno = ENOENT);
                    Message = 'Path does not exist';
                 When (errno = EACCES);
                    Message = 'Permission denied';
                 Other;
                    Message = %Char(errno);
                 EndSl;
             EndIf;
             ExFmt SelectPath;
          EndDo;
          *InLR = *On;
      /End-Free
```

Create this program by using the command:

```
CRTBNDRPG PGM(REDBOOK/API05) SRCFILE(REDBOOK/APISRC)
```

The numbers in the following explanation correspond to the numbered sections in Example 43 on page 73. The main points to note in Example 43 on page 73 are:

1. The H spec includes the QC2LE binding directory for the inclusion of the **__errno()** function.
2. The INDDS keyword maps the display file indicators to a data structure, as opposed to the RPG indicators.
3. All required prototype and named constants definitions are included in STDAPIINFO.
4. A copy member is used to include the standard indicators that are used with a display file.
5. The program loops until F3 is pressed.
6. The **access()** function is called for the requested path with the F_OK (test for existence of a file) access mode. If the return value is zero (successful call), the Message is set to indicate that the requested path exists.
7. Otherwise, **Geterrno()** retrieves the pointer to the errno global variable and the message is set accordingly. Note the use of the named constants for the tests on errno.

Have some fun trying different permutations and combinations of paths!

As with the ILE CEE APIs, the UNIX-Type APIs take a little getting used to. The greatest difficulty is deciphering the documentation.

In Conclusion

You are now ready to tackle the wonderful world of APIs. The following list contains a few last words of advice:

- Before using an API, always scan the Internet to see if someone has invented the wheel before you. There is no point in whiling away hours trying to figure out the permutations and combinations of parameters and formats when someone else has already done the work for you.
- Ensure that the QSYSINC library is installed on your system—it contains prototypes for many APIs, but it is just a pity that they are not all in RPG.
- Create a binding directory for service programs that are not automatically included during program creation, and make the binding directory part of your standard H spec.
- Build up standard copy members for API prototypes and based formats that the APIs use. Many of these formats, like the list header, are used by multiple APIs.
- Encapsulate everything. Place all of your API procedures in service programs where you can easily access them when you need them.

You should have enough to keep you going for quite a while. Have fun!

The team that wrote this IBM Redpaper

This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Susan Gantner has experience that spans over 24 years in the field of application development. She began as a Programmer developing applications for corporations in Atlanta, Georgia, working with a variety of hardware and software platforms. She joined IBM in 1985 and quickly developed a close association with the IBM Rochester Laboratory during the development of the AS/400® system.

Susan worked in Rochester, Minnesota for five years in the AS/400 Technical Support Center. She later moved to the IBM Toronto Software Laboratory to provide technical support for programming languages and AD tools on the AS/400.

Susan left IBM in 1999 to devote more time to teaching and consulting. Her primary emphasis is on enabling customers to take advantage of the latest programming and database technologies on OS/400.

Susan is one of the founding members of System i Developer. She is a regular speaker at COMMON conferences and other technical conferences around the world, and she holds a number of Speaker Excellence medals from COMMON. Susan is a founding partner of System i Developer:

http://www.systemideveloper.com

Jon Paris started his career with IBM midrange systems when he fell in love with the System/38™ while working as a consultant. This love affair ultimately led him to join IBM.

In 1987, Jon was hired by the IBM Toronto Laboratory to work on the S/36 and S/38 COBOL compilers. Subsequently, Jon became involved with the AS/400 and in particular COBOL/400®.

In early 1989 Jon was transferred to the Languages Architecture and Planning Group, with particular responsibility for the COBOL and RPG languages, where he played a major role in the definition of the new RPG IV language and in promoting its use with IBM Business Partners and Users. He was also heavily involved in producing educational and other support materials and services related to other AS/400 programming languages and development tools, such as CODE/400 and VisualAge® for RPG.

Jon left IBM in 1998 to focus on developing and delivering education that was focused on enhancing AS/400 and iSeries® application development skills.

Jon is one of the founding members of System i Developer. He is a frequent speaker at User Groups meetings and conferences around the world, and he holds a number of speaker excellence awards from COMMON. Jon is a founding partner of System i Developer.

Paul Tuohy has worked in the development of IBM midrange applications since the '70s. He was an IT manager for Kodak Ireland Ltd. and Technical Director of Precision Software Ltd. He is currently the CEO of ComCon, which is a midrange consulting company that is based in Dublin, Ireland. He has been teaching and lecturing since the mid-'80s.

Paul is the author of:

- “Re-engineering RPG Legacy Applications”
- “The Programmers Guide to iSeries Navigator”
- The self-teach course “iSeries Navigator for Programmers.”

He writes regular articles for many publications and is one of the quoted industry experts in the IBM Redbooks Publication *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402.

As well as speaking at RPG & DB2® Summits, Paul is also an award winning speaker who speaks regularly at US COMMON and other conferences throughout the world. Paul is a founding partner of System i Developer.

Gary Mullen-Schultz is a certified Consulting IT Specialist at the ITSO, Rochester Center. He leads the team that is responsible for producing RoadRunner and Blue Gene/L™ documentation. Gary and the team focus on i5/OS application development topics, such as PHP, Java, and WebSphere®. He is a Sun™ Certified Java Programmer, Developer, and Architect, and has three issued patents.

Thanks to the following people for their contributions to this project:

Jenifer Servais
International Technical Support Organization, Rochester Center

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Send us your comments in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:
  ibm.com/redbooks
- Send your comments in an e-mail to:
  redbooks@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AS/400®	IBM®	System i™
Blue Gene/L™	iSeries®	System/38™
COBOL/400®	Language Environment®	VisualAge®
DB2®	OS/400®	WebSphere®
Integrated Language Environment®	Redbooks®	
i5/OS®	Redbooks (logo) ®	

The following terms are trademarks of other companies:

Java, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.