

# API Concepts (V5R2)

---

## Table of Contents

- [API Concepts](#)
  - [About application programming interfaces \(APIs\)](#)
  - [API terminology](#)
  - [Language selection considerations](#)
  - [Data types and parameter coding](#)
  - [User space format for list APIs](#)
  - [Error code parameter](#)
  - [Open list information format](#)
  - [Path name format](#)

# API concepts

An application programming interface (API) is defined as a functional interface supplied by the operating system or a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. Some APIs provide the same functions as control language (CL) commands and output file support. Some APIs provide functions that CL commands do not. Most APIs work more quickly and use less system overhead than the CL commands.

APIs allow you to:

- Provide better performance when getting system information or when using system functions provided by control language (CL) commands or file support.
- Use system information and functions that are not available through CL commands.
- Use calls from high-level languages to these interfaces.


For information about using APIs, see the following:


- [About application programming interfaces \(APIs\)](#)
  - [API terminology](#)
  - [Language selection considerations](#)
  - [Data types and parameter coding](#)
  - [User space format for list APIs](#)
  - [Error code parameter](#)
  - [Open list information format](#)
  - [Path name format](#)
-

# About application programming interfaces (APIs)

This topic describes most of the iSeries application programming interfaces (APIs). A few special-purpose APIs are described elsewhere; one example of this is the REXX APIs.

Before using the APIs described in this topic, you should be familiar with the concepts discussed in the [CL](#)

[Programming](#)  book and with the conceptual and guidance information provided in the [System API](#)

[Programming](#)  book on the V5R1 Supplemental Manuals Web site. [Related information](#) provides links to additional information that may be useful to you. On occasion, you may need to refer to other IBM books or topics for more specific information about a particular topic.

## Who should use APIs

The APIs are intended for experienced application programmers who are developing system-level and other OS/400 applications. This topic provides reference information only; it is neither an introduction to the OS/400 licensed program nor a guide to writing OS/400 applications.

## How this information is organized

In the API finder, you can search for APIs by category, by API name, by descriptive name, or by part of the name. You also can search for new APIs, changed APIs, and exit programs.

The API categories are major functional categories, such as backup and recovery, objects, and work management. Within the individual categories, the APIs are organized in alphabetical order as follows:

- By the spelled-out name for the original program model (OPM), the Integrated Language Environment (ILE), and the ILE CEE APIs.
- By the function name for the UNIX-type APIs.

## Code disclaimer information

This topic contains programming examples.

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar functions tailored to your own specific needs.

All sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All programs contained herein are provided to you "AS IS" without any warranties of any kind. The implied warranties of non-infringement, merchantability, and fitness for a particular purpose are expressly disclaimed.

## Compatibility with future releases

In future releases, IBM intends that one of the following will be true:

- If additional input or output parameters are provided for any of the APIs, the new parameters will be placed after the current parameters and will be optional parameters. The existing APIs will continue to work without any changes.
- If an additional data structure is provided, a new format (layout of that data structure) will be created.
- New information may be added to the end of an existing format.

It is IBM's intention that the APIs will continue to work as they originally worked and any existing applications that use the APIs will continue to work without changes. Significant architectural changes, however, may necessitate incompatible changes.

To ensure better compatibility with future releases, you should retrieve and use all of the following when you work with user spaces generated by list APIs:

- Offset values to the list data section
  - Size of the list data section
  - Number of list entries
  - Size of each entry
-

# API terminology

Before using the iSeries APIs, you should be familiar with several terms and special values (not all APIs use every concept). These terms refer to OS/400 objects. The system-recognized identifiers are shown in parentheses:

**binding directory (\*BNDDIR).** An object that contains a list of names of modules and service programs.

**data queue (\*DTAQ).** An object that is used to communicate and store data used by several programs in a job or between jobs.

**module (\*MODULE).** An object that is made up of the output of the compiler.

**program (\*PGM).** A sequence of instructions that a computer can interpret and run. A program can contain one or more modules.

**service program (\*SRVPGM).** An object that packages externally supported callable routines into a separate object.

**user index (\*USRIDX).** An object that provides a specific order for byte data according to the value of the data.

**user queue (\*USRQ).** An object consisting of a list of messages that communicate information to other application programs. Only programming languages that can use machine interface (MI) instructions can access \*USRQ objects.

**user space (\*USRSPC).** An object consisting of a collection of bytes used for storing any user-defined information.

The following special values refer to OS/400 libraries, and you can often use them in API calls in place of specific library names:

**\*ALL.** All libraries, including QSYS, that the user owns or to which the user has read authority.

**\*ALLUSR.** All user-defined libraries to which the user has read authority. \*ALLUSR *includes* libraries that have names starting with the letter Q and that also contain user data. These libraries are:

QDSNX	QSYS2xxxxx	QUSRNOTES
QGPL	QS36F	QUSROND
QGPL38	QUSER38	QUSRPOSGS
QMPGDATA	QUSRADSM	QUSRPOSSA
QMQMDATA	QUSRBRM	QUSRPYMSVR
QMQMPROC	QUSRDIRCL	QUSRRDARS
QPFRDATA	QUSRDIRDB	QUSRSYS
QRCL	QUSRIJS	QUSRVI
QRCLxxxxx	QUSRINFSKR	QUSRVxRxMx
QSYS2		

\*ALLUSR *excludes* System/36 libraries that have names starting with the symbol # and that do not contain user data. Those libraries are:

#CGULIB	#RPGLIB
#COBLIB	#SDALIB
#DFULIB	#SEULIB
#DSULIB	

'xxxxx' is the number of a primary auxiliary storage pool (ASP).

A different library name, in the format QUSRVxRxMx, can be created by the user for each previous release supported by IBM to contain any user commands to be compiled in a CL program for the previous release. For the USRVxRxMx user library, VxRxMx is the version, release, and modification level of a previous release that IBM continues to support.

For more information about QUSRVxRxMx libraries or the \*ALLUSR special value, see [Special values for the SAVLIB command](#) in the Information Center.

**\*CURLIB.** The job's current library. If no current library is specified for the job, the QGPL library is used.

**\*LIBL.** The user and system portions of the job's library list.

**\*USRLIBL.** The user portion of the job's library list.

---

# Language selection considerations

You can use APIs with all the languages available on iSeries business computing systems, except for the ILE APIs. ILE APIs that are implemented as service programs (\*SRVPGM) can be accessed only by ILE languages. In some cases, a program (\*PGM) interface is provided so that non-ILE languages can access the function.

Some APIs also require that particular data types and particular parameter passing conventions be used. The following table shows the languages available on the iSeries system and the data types that they provide.

<i>Language selection considerations -- data types</i>										
Language <sup>1</sup>	Pointers	Binary 2	Binary 4	Character	Zoned Decimal	Packed Decimal	Floating Point	Structures	Single Array	Exception Handling
BASIC (PRPQ 5799-FPK)		X	X	X	X <sup>2</sup>	X <sup>2</sup>	X		X	X
ILE C	X	X	X	X		X <sup>9</sup>	X	X	X	X
VisualAge C++ for OS/400	X	X	X	X		X <sup>10</sup>	X	X	X	X
CL		X <sup>3</sup>	X <sup>3</sup>	X		X		X <sup>4</sup>	X <sup>4</sup>	X
ILE CL	X <sup>5</sup>	X <sup>3</sup>	X <sup>3</sup>	X		X		X <sup>4</sup>	X <sup>4</sup>	X
COBOL	X	X	X	X	X	X		X	X	X <sup>6</sup>
ILE COBOL	X	X	X	X	X	X	X	X	X	X <sup>6</sup>
MI	X	X	X	X	X	X	X	X	X	X
Pascal (PRPQ 5799-FRJ)	X	X	X	X	X <sup>7</sup>	X <sup>7</sup>	X	X	X	X
PL/I (PRPQ 5799-FPJ)	X	X	X	X	X	X	X	X	X	X
REXX				X				X <sup>4</sup>	X <sup>4</sup>	X
RPG		X	X	X	X	X		X	X	X <sup>8</sup>
ILE RPG	X	X	X	X	X	X	X	X	X	X <sup>8</sup>

## Notes:

- 1 You cannot develop Cross System Product (CSP) programs on an iSeries system. You can, however, develop CSP programs on a System/370 system and run them on your iSeries.
- 2 Refer to the CNVRT\$ intrinsic function.
- 3 There is no direct support, but the %BIN function exists on the Change Variable (CHGVAR) CL command to convert to and from binary.
- 4 There is no direct support, but you can use the substring capability to simulate structures and arrays.
- 5 There is no direct support, but pointers passed to a CL program are preserved.
- 6 COBOL and ILE COBOL programs cannot monitor for specific messages, but these programs can define an error handler to run when a program ends because of an error.
- 7 There is no direct support, but you can use extended program model (EPM) conversion routines to convert to and from zoned and packed decimal.
- 8 RPG programs cannot monitor for specific messages, but these programs turn on an error indicator when a called program ends with an error. These programs can define an error handler to run when a program ends because of an error.

- 9 Packed decimal is implemented in ILE C with the decimal() data type.
- 10 Packed decimal is implemented in VisualAge C++ for OS/400 with the Binary Coded Decimal (BCD) class. The BCD class is the C++ implementation of the C-language's decimal(). The BCD object can be used in API calls because it is binary compatible with the decimal() data type.

The following table shows the languages available on the iSeries system and the parameter support that they provide. For more information, see the reference information for the specific programming language that you plan to use.

<i>Language selection considerations -- call conventions</i>			
<b>Language<sup>1</sup></b>	<b>Function Return Values<sup>2</sup></b>	<b>Pass by Reference</b>	<b>Pass by Value</b>
BASIC		X	
ILE C	X	X	X
VisualAge C++ for OS/400	X	X	X
CL		X	
ILE CL		X	
COBOL		X	3
ILE COBOL	X	X	X
MI		X	X
Pascal		X	
PL/I		X	
REXX		X	
RPG		X	
ILE RPG	X	X	X

**Notes:**

- 1 You cannot develop Cross System Product (CSP) programs on an iSeries system. You can, however, develop CSP programs on a System/370 and run them on your iSeries.
- 2 Return values are used by the UNIX-type APIs and the Dynamic Screen Manager (DSM) APIs.
- 3 COBOL provides a by-content phrase, but it does not have the same semantics as ILE C pass-by-value.



# Data types and parameter coding

Data types and parameter coding includes:

- [Data structures and the QSYSINC library](#)
- [Character data](#)
- [Binary data](#)
- [Passing parameters](#)
- [Input and output parameters](#)
- [Offset values and lengths](#)
- [Offset versus displacement considerations for structures](#)
- [Optional parameters](#)
- [Omitted parameters](#)

## Data structures and the QSYSINC library

The QSYSINC (system include) library provides all source includes for APIs shipped with the iSeries system. This optionally installed library is fully supported, which means you can write APARs if you find errors in the includes.

You can install this library by using the GO LICPGM functions of OS/400. Select the Install Licensed Programs option on the Work with Licensed Programs display and the OS/400 System Openness Includes option on the Install Licensed Programs display.

The naming conventions for the includes are the same as either the OPM API or the ILE service program name. If both exist, the include has both names.

<i>Include files shipped with the QSYSINC library</i>			
<b>Operating Environment</b>	<b>Language</b>	<b>File Name</b>	<b>Member Name (Header File)</b>
OPM APIs	ILE C <sup>1</sup>	H	OPM API program name
	RPG	QRPGSRC	OPM API program name or OPM API program name with the letter E replacing the letter Q for members containing array definitions
	ILE RPG	QRPGLESRC	OPM API program name
	COBOL	QLBLSRC	OPM API name
	ILE COBOL	QCBLLSRC	OPM API program name
ILE APIs	ILE C	H	Service program name or API program name <sup>2</sup>
	ILE RPG	QRPGLESRC	Service program name or API program name <sup>2</sup>
	ILE COBOL	QCBLLSRC	Service program name or API program name <sup>2</sup>
UNIX type	ILE C	ARPA	Industry defined
	ILE C	H	Industry defined
	ILE C	NET	Industry defined
	ILE C	NETINET	Industry defined

	ILE C	NETNS	Industry defined
	ILE C	SYS	Industry defined
<b>Notes:</b>			
<ol style="list-style-type: none"> <li><sup>1</sup> CEE ILE APIs are included in this part of the table.</li> <li><sup>2</sup> The API can be either bindable when you use the service program name or callable when you use the API program name.</li> </ol>			

Besides the includes for specific APIs, other includes existing in the QSYSINC library follow:

<i>QLIEPT and QUSEPT</i>	Allow C-language application programs to call OPM APIs directly through the system entry point table
<i>QUSGEN</i>	Defines the generic header for list APIs
<i>QUSEC</i>	Contains the structures for the error code parameter
<i>Qxx</i>	Provides common structures that are used by multiple APIs (where the <i>xx</i> is the component identifier, for example, QMH, QSY, and so forth)

The include files that are shipped with the system define only the fixed portions of the formats. You must define the varying-length fields. The QSYSINC include files are read-only files. If you use a structure that contains one or more varying-length fields, you need to copy the include file to your library and edit your copy. Uncomment the varying-length fields in your copy of the include file, and specify the actual lengths you want. When using a structure as an input to an API, initialize the structure in its entirety (typically to x'00' but refer to the specific API documentation for the correct value) prior to setting specific field values within the structure. This will avoid having to initialize reserved fields by name, as the reserved field name may change in future releases.

Exit programs only have an include if the exit program contains a structure. The QSYSINC member name of these includes is provided in the parameter box for the applicable exit programs.

For development of client-based applications, integrated-file-system symbolic links to QSYSINC openness includes are also provided in the /QIBM/include path.

## Character data

In the API parameter tables, CHAR(\*) represents character data that has:

- A type that is not known, such as character, binary, and so on
- A length that might not be known or is based on another value (for example, a length you specify)

## Binary data

In the API parameter tables, BINARY(2) and BINARY(4) represent numeric data. These parameters must be signed, 2- or 4-byte numeric values with a precision of 15 (halfword) or 31 (fullword) bits and one high-order bit for the sign. Numeric parameters that must be unsigned 4-byte numeric values are explicitly defined as BINARY(4) UNSIGNED.

When you develop applications that use binary values, be aware that some high-level languages allow the definition of binary variables by using precision and not length. For example, an RPG definition of binary

length 4 specifies a precision of 4 digits, which can be stored in a 2-byte binary field. For API BINARY(4) fields, RPG developers should use one of the following:

- Positional notation
- A length of 5 to 9 in order to allocate a 4-byte binary field
- A length of 10 in order to allocate a 4-byte integer field

## Passing parameters

When you call an API, the protocol for passing parameters is to typically pass a space pointer that points to the information being passed. (This is also referred to as pass-by-reference.) This is the convention used by the control language (CL), RPG, and COBOL compilers. Care must be used in those languages that support pass-by-value (such as ILE C) to ensure that these conventions are followed. Refer to the appropriate language documentation for instructions. The parameter passing convention of pass-by-reference can be used in all programming languages. Some of the UNIX-type APIs require pass-by-value parameter passing. VisualAge C++ for OS/400 also supports pass-by-value parameter passing.

## Input and output parameters

API parameters can be used for input or output. Some parameters contain both input and output fields; these are identified as input/output (I/O) parameters in the API parameter tables.

Input parameters and fields are not changed by the API. They have the same value on the return from the API call as they do before the API call. In contrast, output parameters and fields are changed. Any information that an API caller (either an application program or an interactive entry on the display) places in an output parameter or output field before the call will be lost on the return from the call.

## Offset values and lengths

When you are using an API that generates a list into a user space, you should use the offset values and lengths returned by the API in the generic user space header to step through the list instead of specifying what the current version of the API returns. This is because:

- The offset values to the different sections of the user space may change in future releases.
- The length of the entries in the list data section of the user space may change in future releases.

As long as your HLL application program uses the offset values and lengths returned in the generic header of the user space, your program will run in future releases of the OS/400 licensed program.

**Note:** While your application program should use the length returned in the generic header to address subsequent list entries, your application program should only retrieve as many bytes as the application program has allocated storage for.

## Offset versus displacement considerations for structures

You will find the terms offset and or displacement in some of the APIs. For example, the Retrieve Data Queue Message (QMHRDQM) API uses offset; the List Objects (QUSLOBJ) API uses displacement.

An **offset** is the distance from the beginning of an object (user spaces and receiver variables) to the beginning of a particular field. However, a **displacement** is the distance from the beginning of a specific record, block, or structure to the beginning of a particular field.

## Optional parameters

Some of the APIs have optional parameters; the optional parameters form a group. You must either include or exclude the entire group. You cannot use just one of these parameters by itself. In addition, you must include all preceding parameters.

You may call the API in two ways: either with the optional parameters or without the optional parameters.

## Omitted parameters

The Integrated Language Environment (ILE) APIs may have parameters that can be omitted. When these parameters are omitted, you must pass a null pointer.

---

# User space format for list APIs

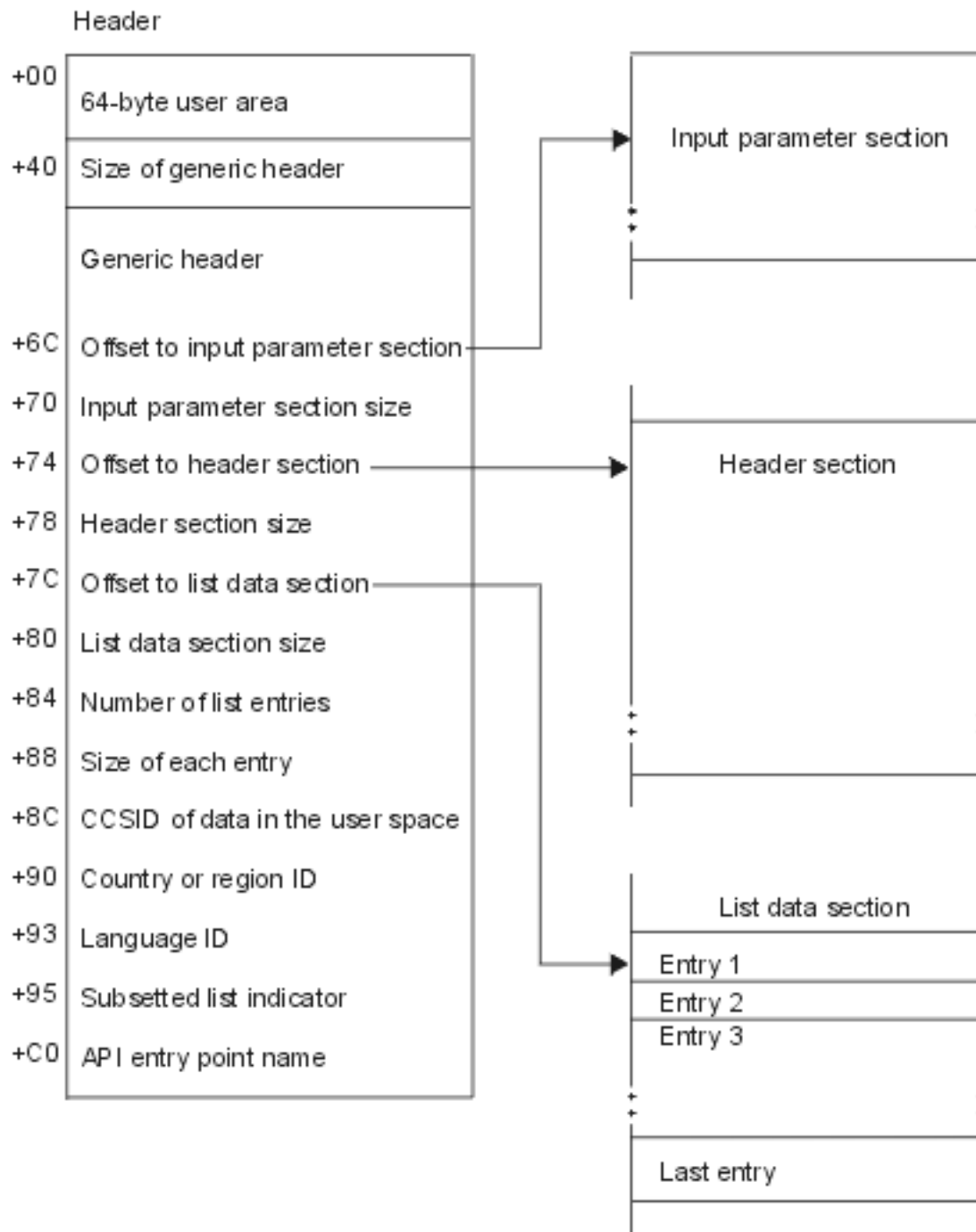
To provide a consistent design and use of the user space (\*USRSPC) objects, the iSeries list APIs use a common data structure. The list APIs are those APIs that generate a list unique to that API. This includes any list API that has a user space parameter, such as the List Spooled Files and List Objects APIs.

Select one of the following for more information:

- [General data structure](#)
- [Common data structure formats](#)
- [Generic header format 0100](#)
- [Generic header format 0300](#)
- [Field descriptions](#)
- [List sections](#)
- [Partial list considerations](#)

## General data structure

The list APIs use the following general data structure:



All offset values are from the beginning of the user space. The offset values for the Dump Object (DMPOBJ) and Dump System Object (DMPSYSOBJ) commands also start at the beginning of the user space. To get the correct starting position for the Change User Space (QUSCHGUS) and Retrieve User Space (QUSRTVUS) APIs, add one to the offset value.

## Common data structure formats

The following tables show 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

## Field descriptions

This topic describes the fields returned in further detail. Field descriptions are in alphabetical order.

**API entry point name.** The name of the ILE bindable API entry point that generated the list.

**API used.** For format 0100, this is the name of the original program model (OPM) API that generated the list. For format 0300, this is a reserved field. See the API entry point name field for the API used.

**CCSID of the data in the list entries.** The coded character set ID for data in the list entries. [»](#)If 0, then the data is not associated with a specific CCSID and should be treated as hexadecimal data. [«](#)

**Country or region ID.** The country or region identifier of the data written to the user space.

**Date and time created.** The date and time when the list was created. The 13 characters are:

- 1* Century, where 0 indicates years 19xx and 1 indicates years 20xx.
- 2-7* The date, in YYMMDD (year, month, day) format.
- 8-13* The time of day, in HHMMSS (hours, minutes, seconds) format.

**Format name.** The name of the format for the list data section.

**Information status.** Whether or not the information is complete and accurate. Possible values are:

- C* Complete and accurate.
- I* Incomplete. The information you received is not accurate or complete.
- P* Partial but accurate. The information you received is accurate, but the API had more information to return than the user space could hold. See [Partial List Considerations](#) for more information about partial lists.

**Language ID.** The language identifier of the data written to the user space.

**Number of list entries.** The number of fixed-length entries in the list data section.

**Offset to (all) section.** The byte offset from the beginning of the user space to the start of the section.

**Reserved.** An ignored field.

**Size of each entry.** The size of each list data section entry, in bytes. All entries are the same size. For formats that return variable length records, this is zero.

**Size of generic header.** The size of the generic header, in bytes. This does not include the size of the user area; refer to [General Data Structure](#) for a diagram showing the user area.

**Size of header section.** The size of the header section, in bytes.

**Size of input parameter section.** The size of the input parameter section, in bytes.

**Size of list data section.** The size of the list data section, in bytes. For formats that return variable length records, this is zero.



**Size of user space used.** The combined size of the user area, generic header, input parameter section, header section, and list data section, in bytes. This determines what is changed in the user space.

**Structure's release and level.** The release and level of the structure. The value of this field is 0100. List APIs put this value into the user space.

**Subsetted list indicator.** A flag that indicates if the data selected from the list API can be stored in that format.

- 0 List is not subsetted; all of the information can be stored in the format.
- 1 List is subsetted. For example, integrated file system names may be longer than the available area in the format. See the API specific documentation for detail.

**User area.** An area within the user space that is provided for the caller to use to communicate system programmer-related information between applications that use the user space.

## List sections

Each list API provides the following sections:

List Section	Contents
Input parameter section	An exact copy of the parameters coded in the call to the API. In general, this section contains all the parameters available.
Header section	Parameter feedback and global information about each object. Some APIs do not use this section; in those cases, the value of the size-of-header-section field is zero.
List data section	The generated list data. All entries in the list section are the same length.

When you retrieve list entry information from a user space, you should use the entry size designated in your application. To get the next entry, use the entry size returned in the generic header. The size of each entry may be padded at the end. If you do not use the entry size, the result may not be valid. For examples of how to process lists, see API [Examples](#).

## Partial list considerations

Some APIs may be able to return more information to the application than fits in a receiver variable or a user space. The information returned is correct, but not complete.

If the list information is not complete, the first item and possibly the second item occur:


- A *P* is returned in the information status field of the generic user space layout; refer to [General Data Structure](#).
- The API supports a continuation handle.

If an indicator of a partial list is returned, the application should call the API again with the continuation handle in the list header section of the API and specify that the list begin with the next entry to be returned.

**Note:** If this is the first time the API is attempting to return information, the continuation handle must be

set to blanks. If the API does not support a continuation handle, you need to call the API again and use more restrictive values for the parameters.

For information about how list APIs make use of user spaces, see User Spaces in the [System API](#)

[Programming](#)  book on the V5R1 Supplemental Manuals Web site. For information about how retrieve APIs make use of receiver variables and to determine whether the returned information is complete, see Receiver Variables in the same book.

---

# Error code parameter

An API error code parameter is common to all of the system APIs.

## Notes:

1. The ILE CEE APIs use feedback codes and conditions. See [OS/400 Messages and the ILE CEE API Feedback Code](#) in the ILE CEE APIs for information about how feedback codes are used with the ILE CEE APIs.
2. The UNIX-type APIs and the National Language Data Conversion APIs use *errno* to report error conditions.

For additional information on the error code parameter, see:

- [Format ERRC0100](#)
- [Format ERRC0200](#)
- [Field descriptions](#)

Most iSeries APIs include an error code parameter to return error codes and exception data to the application. The error code parameter is a variable-length structure that contains the information associated with an error condition. The error code parameter can be one of two variable-length structures, format ERRC0100 or format ERRC0200.

In format ERRC0100, one field in that structure is an INPUT field; it controls whether an exception is returned to the application or the error code structure is filled in with the exception information. When the bytes provided field is greater than or equal to 8, the rest of the error code structure is filled in with the OUTPUT exception information associated with the error. When the bytes provided INPUT field is zero, all other fields are ignored and an exception is returned.

Format ERRC0200 must be used if the API caller wants convertible character (CCHAR) support. Format ERRC0200 contains two INPUT fields. The first field, called the key field, must contain a -1 to use CCHAR support. When the bytes provided field is greater than or equal to 12, the rest of the error code structure is filled in with the OUTPUT exception information associated with the error. When the bytes provided INPUT field is zero, all other fields are ignored and an exception is returned.

For some APIs, the error code parameter is optional. If you do not code the optional error code parameter, the API returns diagnostic and escape messages. If you do code the optional error code parameter, the API returns only escape messages or error codes; it never returns diagnostic messages.

The structure of the error code parameter is as follows. The fields are described in detail after the tables.

**Note:** The error code structures for both formats are provided in the QUSEC include file in the QSYSINC library. Includes exist in the following source physical files: QRPGRSRC, QRPGLSRC, QLBLSRC, QCBLLSRC, and H.

## Format ERRC0100

Offset		Use	Type	Field
Dec	Hex			
0	0	INPUT	BINARY(4)	Bytes provided
4	4	OUTPUT	BINARY(4)	Bytes available

8	8	OUTPUT	CHAR(7)	Exception ID
15	F	OUTPUT	CHAR(1)	Reserved
16	10	OUTPUT	CHAR(*)	Exception data

## Format ERRC0200

Offset		Use	Type	Field
Dec	Hex			
0	0	INPUT	BINARY(4)	Key
4	4	INPUT	BINARY(4)	Bytes provided
8	8	OUTPUT	BINARY(4)	Bytes available
12	C	OUTPUT	CHAR(7)	Exception ID
19	13	OUTPUT	CHAR(1)	Reserved
20	14	OUTPUT	BINARY(4)	CCSID of the CCHAR data
24	18	OUTPUT	BINARY(4)	Offset to the exception data
28	1C	OUTPUT	BINARY(4)	Length of the exception data
		OUTPUT	CHAR(*)	Exception data

## Field descriptions

This topic describes the fields returned in further detail. Field descriptions are in alphabetical order.

**Bytes available.** The length of the error information available to the API to return, in bytes. If this is 0, no error was detected and none of the fields that follow this field in the structure are changed.

**Bytes provided.** The number of bytes that the calling application provides for the error code. If the API caller is using format ERRC0100, the bytes provided must be 0, 8, or more than 8. If more than 32 783 bytes (32KB for exception data plus 16 bytes for other fields) are specified, it is not an error, but only 32 767 bytes (32KB) can be returned in the exception data.

If the API caller is using format ERRC0200, the bytes provided must be 0, 12, or more than 12. If more than 32 799 bytes (32KB for exception data plus 32 bytes for other fields) are specified, it is not an error, but only 32 767 bytes (32KB) can be returned in the exception data.

- 0 If an error occurs, an exception is returned to the application to indicate that the requested function failed.
- >=8 If an error occurs, the space is filled in with the exception information. No exception is returned. This only occurs if format ERRC0100 is used.
- >=12 If an error occurs, the space is filled in with the exception information. No exception is returned. This only occurs if format ERRC0200 is used.

**CCSID of the CCHAR data.** The coded character set identifier (CCSID) of the convertible character (CCHAR) portion of the exception data. The default is 0.

*0* The default job CCSID.

*CCSID* A valid CCSID number. The valid CCSID range is 1 through 65535, but not 65534.

**Exception data.** A variable-length character field that contains the insert data associated with the exception ID.

**Exception ID.** The identifier for the message for the error condition.

**Key.** The key value that enables the message handler error function if CCHAR support is used. This value should be -1 if CCHAR support is expected.

**Length of the exception data.** The length, in bytes, of the exception data returned in the error code.

**Offset to the exception data.** The offset from the beginning of the error code structure to the exception data in the error code structure.

**Reserved.** A 1-byte reserved field.

---

# Open list information format

The format of the open list information is common across many of the open list APIs. The open list APIs return data for use by the process open list APIs. The process open list APIs are located in the Miscellaneous part, whereas the open list APIs can be found in the applicable sections. For example, the Open List of Messages (QGYOLMSG) API is located in the Message Handling part.

The following shows the format of the list information parameter in the open list APIs. For a detailed description of each field, see [Field descriptions](#).

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Total records
4	4	BINARY(4)	Records returned
8	8	CHAR(4)	Request handle
12	C	BINARY(4)	Record length
16	10	CHAR(1)	Information complete indicator
17	11	CHAR(13)	Date and time created
30	1E	CHAR(1)	List status indicator
31	1F	CHAR(1)	Reserved
32	20	BINARY(4)	Length of information returned
36	24	BINARY(4)	First record in receiver variable
40	28	CHAR(40)	Reserved

## Field descriptions

**Date and time created.** The date and time when the list was created. The 13 characters are:

- 1* Century, where 0 indicates years 19xx and 1 indicates years 20xx.
- 2-7* The date, in YYMMDD (year, month, day) format.
- 8-13* The time of day, in HHMMSS (hours, minutes, seconds) format.

**First record in receiver variable.** The number of the first record returned in the receiver variable.

**Information complete indicator.** Whether all requested information has been supplied. Possible values follow:

- C* Complete and accurate information. All of the requested records have been returned in the receiver variable.
- I* Incomplete information. An interruption causes the receiver variable to contain incomplete information.
- P* Partial and accurate information. Partial information is returned when the receiver variable is full and not all of the records requested are returned.

**Length of information returned.** The size, in bytes, of the information that is returned in the receiver variable.

**List status indicator.** The status of building the list. Possible values follow:

- 0 The building of the list is pending.
- 1 The list is in the process of being built.
- 2 The list has been completely built.
- 3 An error occurred when building the list. The next call to the Get List Entries (QGYGTLE) API will cause the error to be signaled to the caller of the QGYGTLE API.
- 4 The list is primed and ready to be built. The list will be built asynchronously by a server job, but the server job has not necessarily started building the list yet.
- 5 Given the current selection criteria and information requested, there is too much data to be returned. The list is incomplete, but data collected to this point is available.

**Record length.** The length of each record of information returned. For variable length records, this value is set to zero. For variable length records, you can obtain the length of individual records from the records themselves.

**Records returned.** The number of records that are returned in the receiver variable. This number is the smallest of the following values:

- The number of records that will fit into the receiver variable.
- The number of records in the list.
- The number of records that are requested.

**Request handle.** The handle of the request that can be used for subsequent requests of information from the list. The handle is valid until the Close List (QGYCLST) API is called to close the list, or until the job ends.

**Note:** This field should be treated as a hexadecimal field. It should not be converted from one CCSID to another, for example, EBCDIC to ASCII, because doing so could result in an unusable value.

**Reserved.** An ignored field.

**Total records.** The total number of records available in the list.

---

# Path name format

The path name format is common across application programming interfaces that work with objects that are supported across file systems. These APIs require a path name to identify the object with which the API will work.

The format of the path name is as follows. For a detailed description of each field, see [Field descriptions](#).

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

## Field descriptions

This section describes the path name format fields in further detail. Field descriptions are in alphabetical order.

**CCSID.** The CCSID (coded character set ID) the path name is in. The possible values follow:

*0* Use the current job default CCSID.

*1-65533* A valid CCSID in this range.

**Country or region ID.** The country or region ID for the path name. The possible values follow:

*X'0000'* Use the current job country or region ID.

*Country or region ID* A valid country or region ID.

**Language ID.** The language ID for the path name. The possible values follow:

*X'000000'* Use the current job language ID.

*Language ID* A valid language ID.

**Length of path name.** The length of the path name in bytes.

**Path name.** Depending on the path type indicator field, this field contains either a pointer to a character string that contains the path name, or a character string that contains the path name.



The path name must be an absolute path name or a relative path name. An absolute path name is a path name that starts with the path name delimiter, usually the slash (/) character. A relative path name is a path name that does not start with the path name delimiter. When a relative name is specified, the API assumes that this path name starts at the current directory of the process that the API is running in.

The dot and dot dot (. ..) directories are valid in the path name. The home directory, generally represented by using the tilde character in the first character position of the path name, is not supported.

**Path name delimiter character.** The delimiter character used between the element names in the path name. This is in the same CCSID as the path name. The most common delimiter is the slash (/) character. If the delimiter is 1 character, the first character of the 2-character field is used.

**Path type indicator.** Whether the path name contains a pointer or is a character string and whether the path name delimiter character is 1 or 2 characters long. The possible values follow:

- 0 The path name is a character string, and the path name delimiter character is 1 character long.
- 1 The path name is a pointer, and the path name delimiter character is 1 character long.
- 2 The path name is a character string, and the path name delimiter character is 2 characters long.
- 3 The path name is a pointer, and the path name delimiter character is 2 characters long.

**Reserved.** A reserved field that must be set to hexadecimal zeros.

---