High-level Language APIs (V5R2)

Table of Contents

High-Level Language APIs

The high-level language APIs communicate with compilers, and the DB2 Universal Database(TM) for iSeries SQL and COBOL/400(R) languages. The high-level language APIs include:

- Application Development Manager/400 APIs

- COBOL/400 APIs

APIs by category

Application Development Manager/400 APIs

The Application Development Manager/400 APIs allow a control language (CL) command such as the Build Part command (BLDPART) to determine, for example, the includes and external references that were used by certain processors when processing a source member. The term **processor** is used in these APIs to mean compiler or preprocessor.

In Application Development Manager/400 terms, a part can be either a source member or an object, such as a file. Refer to the appropriate Application Development Manager/400 publication, as listed in the bibliography, for more information.

If you have an application that can use the information provided by the APIs, you can call these APIs from any high-level programming language. The Application Development Manager/400 feature does not need to be installed on your system for you to use these APIs.

The Get and Set Status APIs are used to query and initialize the build information space that is to contain the Application Development Manager/400 information. The Write and Read Build Information APIs are used to write or read records of build information to and from the space.

For additional information, see Using Application Development Manager/400 APIs.

For information on the different types of records that can be read or written using the Application Development Manager/400 APIs, see:

- Record Types
- Examples of Records Written

The Application Development Manager/400 APIs are:

- Get Space Status (QLYGETS) obtains the status of the space.
- Read Build Information (QLYRDBI) reads one or more records from the space.
- Set Space Status (QLYSETS) sets the status of the space.
- Write Build Information (QLYWRTBI) writes one or more records to the space.

Using Application Development Manager/400 APIs

The following compilers and preprocessors use the the Application Development Manager/400 APIs.

Table 1. Compilers and preprocessors that can be used with the Application Development Manager/400 feature

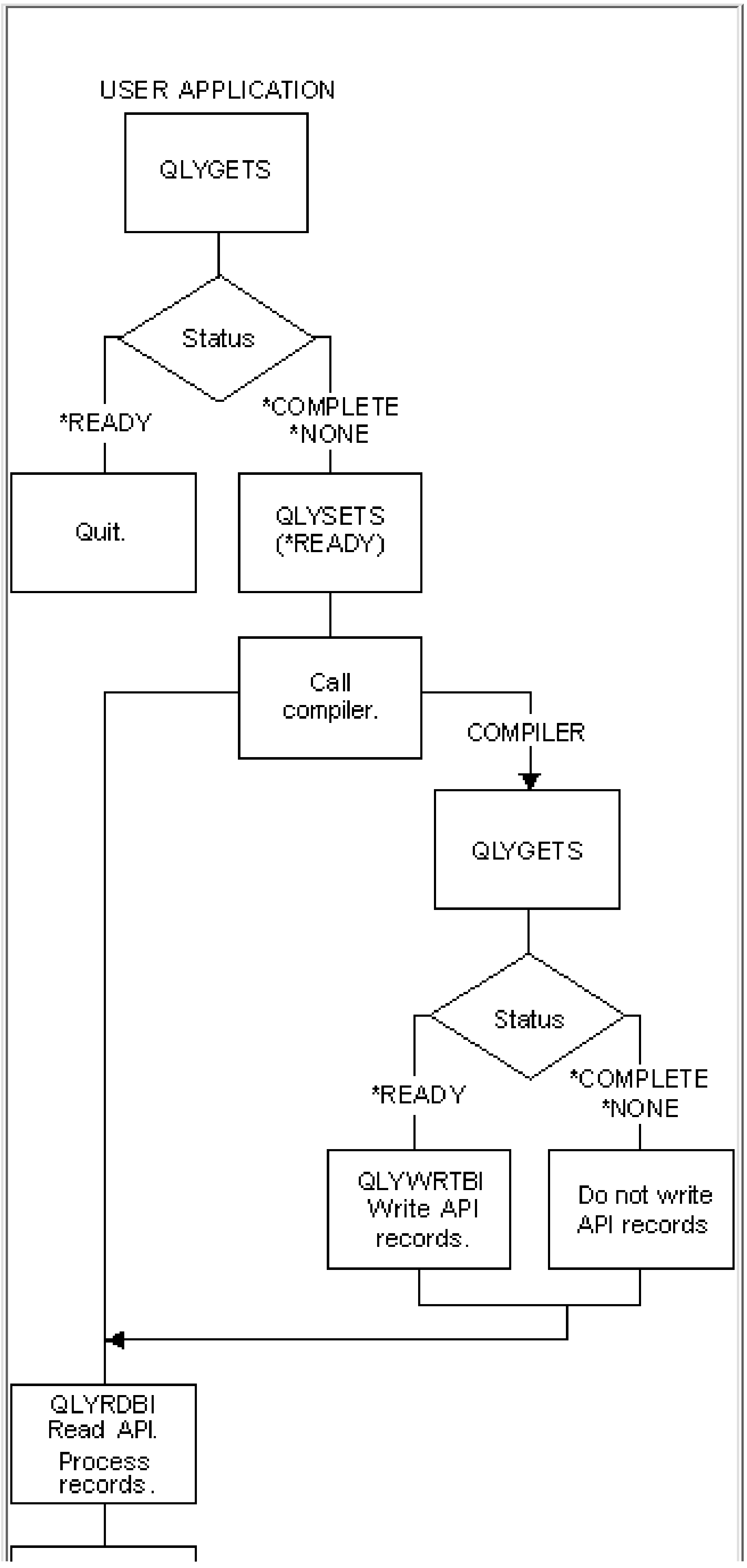
`Compiler/ Preprocessor Language`	**Compiler/Preprocessor OS/400 Command**	**Supported if *PRV is Specified for Target Release**
RPG/400	CRTRPGPGM	Yes
ILE RPG/400	No	
COBOL/400	CRTCBLPGM	Yes
ILE COBOL/400	No	
ILE CL	No	
ILE C	Yes	
CRTPF, CRTLF, CRTDSPF, CRTPRTF, CRTICFF	Not applicable	
CL	CRTCLPGM	Yes
CLD	CRTCLD	Yes
CMD	CRTCMD	Not applicable
CRTSQLRPG, CRTSQLCBL, CRTSQLCI	Yes	
CRTSQLRPGI, CRTSQLCBLI	No	
CRTSRVPGM	CRTSRVPGM	Yes
CRTPGM	CRTPGM	Yes
MENU	CRTMNU TYPE(*UIM)	Not applicable
PNLGRP	CRTPNLGRP	Not applicable

Notes:

1. Default command is used by the `BLDPART` command.
2. Appropriate default compiler command is used based on the part type and the language.

The following diagram shows the proper usage and order in which the APIs should be called.

Figure 1. Overall Application Development Manager/400 API Usage


USER APPLICATION
QLYGETS
Status
*READY
*COMPLETE
*NONE
Quit.
QLYSETS (*READY)
Call compiler.
COMPILER
QLYGETS
Status
*READY
*COMPLETE
*NONE
QLYWRTBI Write API records.
Do not write API records
QLYRDBI Read API.
Process records.

QLYGETS should be called by the application or compiler before calling the other three APIs: QLYSETS, QLYWRTBI, and QLYRDBI to verify that the space is available for use.

The following table describes the API space status values that can be received by calling the QLYGETS API, and the action that should be taken by the application or compiler that is calling the API.

Table 2. API Space Status

Status	**Application**	**Compiler**
*COMPLETE	The space is available for use. Call QLYSETS to set to *READY.	Do not write API records.
*NONE	The space does not exist. The application calls QLYSETS to create and set the space to *READY.	Do not write API records.
*READY	The space is in use by a compiler. The other APIs should not be called.	The space is available for writing.

Compilers use the APIs to write to the space. Applications use the APIs to read from the space.

Note: Unpredictable results can occur when the APIs are not properly used or are used in the incorrect order.

Calling multiple API-supporting compilers simultaneously in a single interactive session (one possible way of doing this is by pressing the Attention key and then command key F9 to get to the command line) may cause unpredictable results. The compiler can fail, for example, or incorrect or incomplete information can be put in the work space.

Record Types

This section describes the information contained in all the different record types. Typically a compiler writes records and an application reads them.

Names, field types and other information passed through the different record types are *not* validated and no authority is checked by QLYWRTBI. The QLYWRTBI API assumes that all that validation and checking has been done.

There are the following record types:

- Processor member start record
- Processor object start record
- Normal processor end record
- Normal processor end call next record
- Normal multiple end record
- Abnormal processor end record
- Include record
- File reference record
- Module reference record
- Service program reference record
- Bind directory reference record
- Record format reference record
- Field reference record
- Message reference record
- External reference error record
- Object already exists error record
- Start of new program record

The following table shows the records that can be written by each compiler.

All fields where information is not available to put in these records should be filled with blanks.

The following is true for the *Library specified* fields for all records and compilers:

- When *CURLIB is specified for the *Library specified* fields, *CURLIB is passed.
- When *LIBL is specified for the *Library specified* fields, or implied by not being specified, *LIBL is passed.

Notes and restrictions are explained in the footnotes following the tables.

Record Types and Processors (Part 1)

Record Type	**Record ID**	**RPG/400: CRTRPGPGM**	**COBOL/400: CRTCBLPGM**	**CLD: CRTCLD**	**DDS: CRTPF CRTLF CRTDSPF CRTICFF CRTPRTF**	**CL: CRTCLPGM**	**CMD: CRTCMD**
Processor member start	'01'	X(1, 3)	X	X(1, 3)	X	X(1, 3, 5)	X(1)
Processor object start	'50'						
Normal processor end	'20'	X	X	X	X	X(5)	X

Normal processor end call next	'21'						
Normal multiple end record	'65'						
Abnormal processor end	'30'	X	X	X	X	X(5)	X
Include	'02'	X(11)	X				
File reference	'03'	X	X		X	X(1, 5)	
Module reference	'55'						
Service program reference	'60'						
Bind directory reference	'75'						
Record format reference	'04'	X	X		X	X(1, 5)	
Field reference	'05'				X(2)		
Message reference	'06'				X(2, 9)		X(1, 2, 6, 9)
External reference error	'15'	X(10)	X		X(10)	X(1, 4, 5)	
Object already exists error	'16'				X		
Start of new program	'40'		X(20)				

Record Types and Processors (Part 2)

Record Type	**Record ID**	**DB2 UDB for iSeries: CRTSQLRPG CRTSQLCBL**	**ILE RPG/400: CRTRPGMOD CRTBNDRPG**	**ILE COBOL/400: CRTCBLMOD CRTBNDCBL**	**ILE C: CRTCMOD CRTBNDC**	**ILE CL: CRTCLMOD CRTBNDCL**	**ILE DB2 UDB for iSeries: CRTSQLRPGI CRTSQLCBLI CRTSQLCI**
Processor member start	'01'	X(1)	X(1, 3)	X	X(3)	X(1, 3)	X(1)
Processor object start	'50'						

Normal processor end	'20'	X	X	X	X	X	X
Normal processor end call next	'21'	X	X(14)	X(14)	X(14)	X(14)	X
Normal multiple end record	'65'						
Abnormal processor end	'30'	X	X	X	X	X	X
Include	'02'	X(1, 7)	X(11)	X	X(8, 12)		X(1, 7)
File reference	'03'	X(1)	X	X	X	X(1)	X(1)
Module reference	'55'						
Service program reference	'60'						
Bind directory reference	'75'						X
Record format reference	'04'	X(1)	X	X	X	X(1)	X(1)
Field reference	'05'						
Message reference	'06'						
External reference error	'15'	X(1)	X(10)	X	X(10, 13)	X(1, 4)	X(1)
Object already exists error	'16'						
Start of new program	'40'			X(20)			X(20)

Record Types and Processors (Part 3)

Record Type	**Record ID**	**ILE SRVPGM: CRTSRVPGM**	**ILE CRTPGM**	**UIM: CRTPNLGRP**	**CRTMNU**	**UDT: SYSTYPE(*NONE)**	**UDT: member**
Processor member start	'01'	X(18)		X	X(17)		X
Processor object start	'50'		X(16)			X(19)	

Normal processor end	'20'	X	X	X	X		X
Normal processor end call next	'21'						X
Normal multiple end record	'65'					X(19)	
Abnormal processor end	'30'	X	X	X	X	X(19)	X
Include	'02'			X	X		X
File reference	'03'						X
Module reference	'55'	X	X				
Service program reference	'60'	X	X				
Bind directory reference	'75'	X	X				
Record format reference	'04'						X
Field reference	'05'						X
Message reference	'06'			X	X		X
External reference error	'15'	X(15)	X(15)	X	X		X
Object already exists error	'16'						X
Start of new program	'40'						X

Notes and Restrictions for the Above Tables:

1. If *CURLIB is specified for the *Library specified* fields (this includes the *Source library specified* field on the Processor member start record), the resolved library name is passed instead of *CURLIB.

2. If *LIBL is specified for the *Library specified* fields, or implied by not being specified, the resolved library name is passed instead of *LIBL.

3. If *CURLIB is specified for the *Target library* field, the resolved library name is passed instead of *CURLIB.

4. For most *Used* fields, when a file being referenced on the DCLF command cannot be found, CL puts blanks in this field. There is no actual file or library name when the file is not found.

5. For all fields marked *Reserved*, CL initializes them to hex zeros. However, fields that are not reserved are set to blanks when they do not apply and are defined as characters. For example, *Target member* on the Processor member start record does not have meaning for the CL compiler and is initialized to blanks.

6. Message reference records are written only for messages specified on the PROMPT parameter of the PARM, ELEM, or QUAL command definition statement.

7. The SQL compilers do not write include records for the following statements:

   ```
   EXEC SQL INCLUDE SQLCA
   EXEC SQL INCLUDE SQLDA
   ```

   These statements are not true includes in the sense that the SQL compiler does not read source from another member or source file.
8. The ILE C compiler does not write API Include records for system include files. File names enclosed in angle brackets, (< ... >), designate system include files. File names enclosed in double quotation marks, (" ... "), designate user include files.

9. The *Message file used* and *Library used* fields are always blank.

10. If *LIBL is specified in the source, or implied by not being specified (*Library specified* is *LIBL), the *Library used* field is set to *LIBL because no specific library can be determined if the file is not found in the library list.

11. The RPG/400 compiler puts *LIBL in the *Library specified* field if it is not already specified, and QRPGSRC in the *File specified* field if it is not already specified.

    The ILE RPG/400 compiler puts *LIBL in the *Library specified* field if it is not already specified, and QRPGLESRC in the *File specified* field if it is not already specified.
12. The *Library specified* field is the resolved library name if the library name is not already specified. The *Include file specified* field contains the resolved file name if the file name is not already specified.

13. If *CURLIB is specified in the source (*Library specified* is *CURLIB), the *Library used* field is set to *CURLIB because no specific library can be determined if the file is not found in the library list.

14. This record is written only by the CRTBNDxxx commands.

15. This record is written only when a SRVPGM or MODULE does not exist, and this causes the compilation to fail.

16. The object fields in this record refer to the ENTMOD parameter for the CRTPGM command.

17. CRTMNU only writes records when TYPE(*UIM) is specified.

18. The source used fields contain the same information as the source specified fields.

19. User-defined types are part types that the user created and not the part types made available with the Application Development Manager feature. See the ADTS/400: Application Development Manager Self-Study Guide book on the V5R1 Supplemental Manuals Web site for more information on creating and using user-defined part types.

20. Any COBOL/400 source may contain more than one program.

Processor member start record

This, or the Processor object start record, must be the first record that is passed by the compiler or preprocessor on its first call to the QLYWRTBI API. Its purpose is to identify the source that is being compiled, and also to describe the expected output object, if any.

Note: This record was previously called the **processor start** record, but the format remains the same.

The Processor member start record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Processor command
18	12	CHAR(10)	Source object name specified
28	1C	CHAR(10)	Source library name specified
38	26	CHAR(7)	Source object type
45	2D	CHAR(10)	Source member name specified
55	37	CHAR(10)	Source object name used
65	41	CHAR(10)	Source library name used
75	4B	CHAR(10)	Source member name used
85	55	CHAR(10)	Target object name specified
95	5F	CHAR(10)	Target library name specified
105	69	CHAR(7)	Target object type
112	70	CHAR(10)	Target member name specified
122	7A	CHAR(2)	Reserved

Processors for which this record type applies

All compilers and preprocessors listed in Record Types and Processors (Part 1) except CRTPGM, and the processor processing the user-defined types added with SYSTYPE(*NONE) on the ADDADMTYPE command.

Field Descriptions

Processor command. The compiler or preprocessor that wrote this record, for example, CRTRPGPGM.

Record length. The length of this record is 124.

Record type. The type of this record is '01'.

Reserved. An ignored field.

Source library name used. The actual name of the library that was used. The library name could be different from the specified library name because *LIBL or *CURLIB was specified, or an override was used. This field contains the name the library resolves to.

Source library name specified. The library name of the source file specified on the compiler or preprocessor command.

Source member name used. The actual name of the source member that was used. This field is required, even if the two member names are the same.

Source member name specified. The source member name specified on the compiler or preprocessor command.

Source object name used. The actual name of the object that was used. The object name could be different from the specified object name if an override was used.

Source object name specified. The object name specified on the compiler or preprocessor command.

Source object type. The OS/400 type of the source object (for example, *FILE).

Target library name specified. The library of the target object specified on the compiler or preprocessor command.

Target member name specified. The name of the member to be created, if applicable, specified on the compiler or preprocessor command.

Target object name specified. The name of the object to be created, called the target object, specified on the compiler or preprocessor command. The actual name of the object that was created is passed through the Normal processor end record. (See Normal processor end record.)

Target object type. The OS/400 type of the object to be created (for example, *FILE).

Processor object start record

This, or the Processor member start record, must be the first record that is passed by the compiler or preprocessor on its first call to the QLYWRTBI API. Its purpose is to identify the object that is being processed, and also to describe the expected output object, or, for user-defined types, the expected location of the output members, if any.

User-defined types added with SYSTYPE(*NONE) on the ADDADMTYPE command must write this record before any other record.

The Processor object start record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Processor command
18	12	CHAR(10)	Object name specified
28	1C	CHAR(10)	Object library name specified
38	26	CHAR(7)	Object type specified
45	2D	CHAR(10)	Object name used
55	37	CHAR(10)	Object library name used
65	41	CHAR(7)	Object type used
72	48	CHAR(10)	Target object name specified
82	52	CHAR(10)	Target object library name specified
92	5C	CHAR(7)	Target object type specified
99	63	CHAR(1)	Reserved

Processors for which this record type applies

CRTPGM and the processor processing the user-defined types added with SYSTYPE(*NONE) on the ADDADMTYPE command.

Field Descriptions

Object library name specified. The library name of the object specified on the compiler or preprocessor command. If the object type specified is a user-defined type with SYSTYPE(*NONE), the library name specified should be the group library name.

Object library name used. The actual name of the library that the object was found in. The library name could be different from the specified library name because, for example, *LIBL or *CURLIB was specified. This field contains the name the library resolves to.

Object name specified. The object name specified on the command. If the object type specified is a user-defined type with SYSTYPE(*NONE), the object name specified should be the part name.

Object name used. The actual name of the object that was used. The object name could be different from the specified object name if an override was used.

Object type specified. The object type specified on the command. For user-defined types this must be left blank. If the object type specified is a user-defined type with SYSTYPE(*NONE), the object type specified should be the part type.

Object type used. The actual type of the object used. For example, *MODULE. For user-defined types this can be left blank.

Processor command. The compiler or preprocessor that wrote this record, for example, CRTPGM.

Record length. The length of this record is 100.

Record type. The type of this record is '50'.

Reserved. An ignored field.

Target object library name specified. The library of the target object specified on the command. For user-defined types, the library where the output members are created, as specified on the command.

Target object name specified. The name of the object to be created, or modified as specified on the command. For user-defined types this can be left blank.

Target object type specified. The type of the object to be created. For example, *PGM. The actual name of the object that was created is passed through the Normal processor end record. (See Normal processor end record.) For user-defined types, the names of the output members are passed through the Normal multiple end record. For user-defined types this value must be *MBR.

Normal processor end record

This is the last record passed by the compiler or preprocessor to indicate that processing ended successfully.

The Normal processor end record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Object name created
18	12	CHAR(10)	Library
28	1C	CHAR(7)	Object type
35	23	CHAR(10)	Member
45	2D	CHAR(7)	Message identifier

Processors for which this record type applies

All compilers and preprocessors listed in Record Types and Processors (Part 1), except the processor processing the user-defined types added with SYSTYPE(*NONE) on the ADDADMTYPE command.

Field Descriptions

Library name. The library where the object was created.

Member name. The name of the member created, if applicable.

Message identifier. The message identification of the completion message.

Object name created. The object created by the compiler or preprocessor. If an object is not created, this field stores the value of '*NONE'.

Object type. The type of object created.

Record length. The length of this record is 52.

Record type. The type of this record is '20'.

Reserved. An ignored field.

Normal processor end call next record

When a preprocessor successfully creates an object or a member and needs to call another compiler or preprocessor, it should pass this record instead of passing the Normal processor end record as the final record. For example, if the CRTSQLCI command is entered with OPTION(*GEN), and the member is created successfully, the last record written by CRTSQLCI is the Normal processor end call next record. The preprocessor then calls the CRTBNDC command that eventually writes the Normal or Abnormal processor end record.

The Normal processor end call next record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Object name
18	12	CHAR(10)	Library name
28	1C	CHAR(7)	Object type
35	23	CHAR(10)	Member name
45	2D	CHAR(7)	Message identifier

Processors for which this record type applies

CRTSQLRPG	CRTSQLCBL	CRTSQLRPGI	CRTSQLCBLI
CRTBNDRPG	CRTBNDCBL	CRTBNDC	CRTBNDCL
CRTSQLCI when OPTION(*GEN) is specified	processor processing the user-defined types represented as members		

Field Descriptions

Library name. The library where the object was created.

Member name. The name of the member created, if applicable.

Message identifier. The message identification of the completion message.

Object name. The name of the object created.

Object type. The type of object created.

Record length. The length of this record is 52.

Record type. The type of this record is '21'.

Reserved. An ignored field.

Normal multiple end record

This is the last record passed by a user-defined type added with SYSTYPE(*NONE) on the ADDADMTYPE command. It identifies Normal multiple end processing of **all** the output members. One Normal multiple end record is written per member generated. The Normal processor end record should not be written.

Note: It is possible that the processor generated 10 members on the last build, and because of a change, now needs to regenerate just 2 of those members. For the build process to preserve the relationships to the remaining 8 members, the processor must write all members to the API, regardless of whether the member was actually regenerated. The build process ignores those parts (members) that have either not changed (because the processor did not regenerate them), or do not exist (because the processor did not generate them, and they may exist higher in the hierarchy).

The Normal multiple end record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Library
18	12	CHAR(10)	File name created
28	1C	CHAR(10)	Member
38	26	CHAR(32)	Part type
70	46	CHAR(32)	Part language
102	66	CHAR(22)	Reserved

Processors for which this record type applies

The processor processing the user-defined types added with SYSTYPE(*NONE) on the ADDADMTYPE command.

Field Descriptions

File name created. The file name that was created or used to hold the member.

Library. The library where the member was created.

Member. The name of the member created.

Part language. The language of the part to represent this member.

Part type. The type of the part to represent this member.

Record length. The length of this record is 124.

Record type. The type of this record is '65'.

Reserved. An ignored field.

Abnormal processor end record

This is the last record passed if the compiler or preprocessor fails because of an error. For example, an object or a member was not created because of compile errors, or REPLACE(*NO) was specified on the command and the object existed.

If the command failed because an external reference to a file, message file, module, bind directory or service program could not be found, the command passes the External reference error record before passing this one. See External reference error record for more information on this record.

The Abnormal processor end record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(7)	Message identifier
15	F	CHAR(1)	Reserved

Processors for which this record type applies

All compilers and preprocessors listed in Record Types and Processors (Part 1).

Field Descriptions

Message identifier. The message identification of the completion message.

Record length. The length of this record is 16.

Record type. The type of this record is '30'.

Reserved. An ignored field.

Include record

This record is passed when the compiler or preprocessor processes an include. An **include** statement is a statement that causes the compiler to replace the include statement with the contents of the specified header or file. If the include is not found, the compiler or preprocessor passes the Abnormal processor end record.

The Include record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	BINARY(4)	Nesting level
12	C	CHAR(10)	Include file name specified
22	16	CHAR(10)	Include file library name specified
32	20	CHAR(10)	Include file member name specified
42	2A	CHAR(7)	Object type
49	31	CHAR(10)	Include file name used
59	3B	CHAR(10)	Include file library name used
69	45	CHAR(10)	Include file member name used
79	4F	CHAR(1)	Reserved

Processors for which this record type applies

CRTRPGPGM	CRTCBLPGM	CRTRPGMOD	CRTBNDRPG
CRTCBLMOD	CRTBNDCBL	CRTCMOD	CRTBNDC
CRTSQLRPGI	CRTSQLCBLI	CRTSQLCI	CRTPNLGRP
CRTMNU	processor processing the user-defined types represented as members		

Field Descriptions

Include file used. The actual name of the include file that was used. For example, the default include file used by the compiler and implied in the source, or the file different from the one specified in the source as a result of an override. This name must always be filled in.

Include file specified. The name of the file that contains the include. This is the name specified in the source (if the include was file qualified), otherwise it is blank.

Include file library used. The name of the actual library that contains the include file that was used (for example, a specific library name instead of *CURLIB or *LIBL, as specified in the source, or a library different from the one specified in the source, as a result of an override).

Include file library specified. The name of the library where the include file resides, as specified in the source (if the include was library qualified), otherwise it is blank.

Include file member used. The actual name of the source member containing the include that was used. This name must always be filled in.

Include file member specified. The name of the source member containing the include, as specified in the source.

Nesting level. The level of nesting of the include. Includes found in the root source have a nesting level of 1, includes found in level 1 have a nesting level of 2 and so on.

Object type. The object type of the object containing the include, for example *FILE.

Record length. The length of this record is 80.

Record type. The type of this record is '02'.

Reserved. An ignored field.

The nesting level should be indicated even by those compilers that do not allow include nesting. In that case, the nesting level passed should be equal to 1.

File reference record

This record is passed when the compiler or preprocessor encounters a reference to an externally described file but not its record format or field.

For example, a reference is made in DDS source using the PFILE or JFILE keywords. Another example is when a compiler or preprocessor copies *all* the record format declares from a file. This is not considered to be a dependency on any specific record format and is treated as a dependency on the file, so this record must be passed, not the Record format reference records for all the individual record formats.

The File reference record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	File name specified
18	12	CHAR(10)	File library name specified
28	1C	CHAR(1)	Based on indicator
29	1D	CHAR(10)	File name used
39	27	CHAR(10)	File library name used
49	31	CHAR(3)	Reserved
52	34	BINARY(4)	Nesting level

Processors for which this record type applies

CRTRPGPGM	CRTCBLPGM	CRTPF	CRTLF
CRTDSPF	CRTICFF	CRTPRTF	CRTCLPGM
CRTSQLRPG	CRTSQLCBL	CRTRPGMOD	CRTBNDRPG
CRTCBLMOD	CRTBNDCBL	CRTCMOD	CRTBNDC
CRTCLMOD	CRTBNDCL	CRTSQLRPGI	CRTSQLCBLI

CRTSQLCI	processor processing the user-defined types represented as members

Field Descriptions

Based on indicator. Indicates whether the referenced file is used to base another file on. Possible values are N (no) and Y (yes).

File name used. The name of the actual file that was referenced. This name must always be filled in.

File name specified. The name of the file referenced, as specified in the source.

File library name used. The name of the actual library that contains the file that was referenced. The library name could be different from the specified library name because *LIBL or *CURLIB was specified, or an override was used.

File library name specified. The name of the library of the file referenced, as specified in the source.

Nesting level. If this file reference is made within an include, this field has value of N + 1, where N is the nesting level of the include. Otherwise, the value of this field is 1.

Record length. The length of this record is 56.

Record type. The type of this record is '03'.

Reserved. An ignored field.

Module reference record

This record is passed when a module is successfully referenced by a processor. This record is not to be written for the ENTMOD module, on the CRTPGM command.

The Module reference record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Module name specified
18	12	CHAR(10)	Module library name specified
28	1C	CHAR(10)	Module name used
38	26	CHAR(10)	Module library name used

Processors for which this record type applies

CRTSRVPGM and CRTPGM.

Field Descriptions

Module name used. The name of the actual module that was referenced. This name must always be filled in.

Module name specified. The name of the module referenced, as specified on the command, or in the bind directory.

Module library name used. The name of the actual library that contains the module that was referenced. The library name could be different from the specified library name because *LIBL or *CURLIB was specified.

Module library name specified. The name of the library of the module referenced, as specified on the command, or in the bind directory.

Record length. The length of this record is 92.

Record type. The type of this record is '55'.

Reserved. An ignored field.

Service program reference record

This record is passed when a service program is successfully referenced by a processor.

The Service program reference record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Service program name specified
18	12	CHAR(10)	Service program library name specified
28	1C	CHAR(10)	Service program name used
38	26	CHAR(10)	Service program library name used
48	30	CHAR(16)	Service program signature used

Processors for which this record type applies

CRTSRVPGM and CRTPGM.

Field Descriptions

Record length. The length of this record is 64.

Record type. The type of this record is '60'.

Service program name used. The name of the actual service program that was referenced. This name must always be filled in.

Service program name specified. The name of the service program as specified on the command.

Service program library name used. The name of the actual library that contains the service program that was referenced. The library name could be different from the specified library name because *LIBL or *CURLIB was specified.

Service program library name specified. The name of the library of the service program referenced, as specified on the command.

Service program signature used. The current signature of the service program used.

Bind directory reference record

This record is passed when a module is successfully referenced by a processor. This record is not to be written for the ENTMOD module, on the CRTPGM command.

The Bind directory reference record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Bind directory name specified
18	12	CHAR(10)	Bind directory library name specified
28	1C	CHAR(10)	Bind directory name used
38	26	CHAR(10)	Bind directory library name used

Processors for which this record type applies

CRTSRVPGM and CRTPGM.

Field Descriptions

Bind directory name used. The name of the actual bind directory that was referenced. This name must always be filled in.

Bind directory name specified. The name of the bind directory referenced, as specified on the command.

Bind directory library name used. The name of the actual library that contains the bind directory that was referenced. The library name could be different from the specified library name because *LIBL or *CURLIB was specified.

Bind directory library name specified. The name of the library of the bind directory referenced, as specified on the command.

Record length. The length of this record is 48.

Record type. The type of this record is '75'.

Reserved. An ignored field.

Record format reference record

This record is passed when the compiler or preprocessor encounters a reference to a record format of an externally described file (but not to any single field). For example, a reference is made in DDS source using the FORMAT keyword or in the RPG, COBOL, CL, DB2 UDB for iSeries SQL, ILE RPG, ILE COBOL, ILE CL, or ILE C processors whenever a declaration of a record format structure from a DDS-described file is generated by the compiler or preprocessor.

The Record format reference record has the following format:

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	File name specified
18	12	CHAR(10)	File library name specified
28	1C	CHAR(10)	Record format name
38	26	CHAR(13)	Record format level ID
51	33	CHAR(10)	File name used
61	3D	CHAR(10)	File library name used
71	47	CHAR(1)	Reserved
72	48	BINARY(4)	Nesting level

Processors for which this record type is applicable

CRTRPGPGM	CRTCBLPGM	CRTPF	CRTLF
CRTDSPF	CRTICFF	CRTPRTF	CRTCLPGM
CRTSQLRPG	CRTSQLCBL	CRTRPGMOD	CRTBNDRPG
CRTCBLMOD	CRTBNDCBL	CRTCMOD	CRTBNDC
CRTCLMOD	CRTBNDCL	CRTSQLRPGI	CRTSQLCBLI
CRTSQLCI	processor processing the user-defined types represented as members		

Field Descriptions

File name used. The name of the actual file that was referenced. This name must always be filled in.

File name specified. The name of the file being referenced, as specified in the source.

File library name used. The name of the actual library that contains the file that was referenced. The library name could be different from the specified library name because *LIBL or *CURLIB was specified, or an override was used. This field contains the name the library resolves to.

File library name specified. The name of the library of the file being referenced, as specified in the source.

Nesting level. If this record format reference is made within an include, this field has value of N + 1, where N is the nesting level of the include. Otherwise, the value of this field is 1.

Record format level ID. The level ID of the record format referenced.

Record format name. The name of the record format referenced.

Record length. The length of this record is 76.

Record type. The type of this record is '04'.

Reserved. An ignored field.

Field reference record

This record is passed when the compiler or preprocessor encounters a reference to a field in an externally described file. For example, a reference is made in DDS source using the REF and REFFLD keywords.

The Field reference record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	File name specified
18	12	CHAR(10)	File library name specified
28	1C	CHAR(10)	Record format name
38	26	CHAR(13)	Record format level ID
51	33	CHAR(10)	Field
61	3D	CHAR(3)	Reserved
64	40	BINARY(4)	Field length
68	44	BINARY(4)	Decimal positions
72	48	CHAR(1)	Data type
73	49	CHAR(1)	Fixed/variable length indicator
74	4A	CHAR(10)	File name used
84	54	CHAR(10)	File library name used
94	5E	CHAR(2)	Reserved

Processors for which this record type applies

CRTPF	CRTLF	CRTDSPF	CRTICFF
CRTPRTF	processor processing the user-defined types represented as members		

Field Descriptions

Data type. The field data type in DDS. For example, P, S, B, F, A, or H.

Decimal positions. The number of decimal positions if the field is numeric, otherwise 0.

Field. The name of the referenced field.

Field length. The length of the field in bytes. If the field is a variable-length field, the maximum length should be passed.

File name used. The name of the actual file that was referenced. This name must always be filled in.

File name specified. The name of the file being referenced, as specified in the source.

Fixed/variable length indicator. Contains F if the field is of fixed length, or V if variable length.

File library name used. The name of the actual library that contains the file that was referenced.

File library name specified. The name of the library of the file being referenced, as specified in the source.

Record format level ID. The level ID of the record format referenced.

Record format name. The name of the record format referenced.

Record length. The length of this record is 96.

Record type. The type of this record is '05'.

Reserved. An ignored field.

Message reference record

This record is passed when the compiler encounters a reference to a message ID in a message file. For example, a reference is made in DDS source using the MSGCON keyword.

The Message reference record has the following format:

Offset		Type	Field
Dec	**Hex**		
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(7)	Message identifier
15	F	CHAR(10)	Message file name specified
25	19	CHAR(10)	Message file library name specified
35	23	CHAR(10)	Message file name used
45	2D	CHAR(10)	Message file library name used
55	37	CHAR(1)	Reserved
56	38	BINARY(4)	Nesting Level

Processors for which this record type applies

CRTPF	CRTLF	CRTDSPF	CRTPRTF
CRTICFF	CRTCMD	CRTPNLGRP	CRTMNU
processor processing the user-defined types represented as members			

Field Descriptions

Message file library used. The name of the actual library that contains the message file. This may be *CURLIB or *LIBL if the compiler does not resolve to the library name.

Message file library specified. The name of the library that contains the message file, as specified in the source.

Message file name used. The name of the actual message file that was referenced. This name must always be filled in.

Message file name specified. The name of the message file referenced, as specified in the source.

Message identifier. The message ID referenced.

Nesting Level. The level of nesting of the MSGF. MSGFs referenced in the root source have a nesting level of 1, MSGFs found in level 1 have a nesting level of 2 and so on.

Record length. The length of this record is 60.

Record type. The type of this record is '06'.

Reserved. An ignored field.

External reference error record

This record is passed when processing fails because a referenced object, such as a file, message file, module, bind directory or service program cannot be found. This record does **not apply to includes**.

After passing one or more of these records, the compiler or preprocessor also passes the Abnormal processor end record (see Abnormal processor end record).

The External reference error record has the following format:

Offset		Type	Field
Dec	**Hex**		
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Object name specified
18	12	CHAR(10)	Object library name specified
28	1C	CHAR(7)	Object type
35	23	CHAR(10)	Object name used
45	2D	CHAR(10)	Object library name used
55	37	CHAR(1)	Based on indicator

Processors for which this record type applies

CRTRPGPGM	CRTCBLPGM	CRTPF	CRTLF
CRTDSPF	CRTICFF	CRTPRTF	CRTCLPGM
CRTSQLRPG	CRTSQLCBL	CRTRPGMOD	CRTBNDRPG
CRTCBLMOD	CRTBNDCBL	CRTCMOD	CRTBNDC
CRTCLMOD	CRTBNDCL	CRTSQLRPGI	CRTSQLCBLI
CRTSQLCI	CRTSRVPGM	CRTPGM	CRTPNLGRP
CRTMNU	processor processing the user-defined types represented as members		

Field Descriptions

Based on indicator. Whether the referenced file is used to base another file on. Possible values are N (no) and Y (yes). This field is used by the CRTLF processor.

Object library name used. The actual name of the library that contains the object that was referenced.

Object library name specified. The name of the library that contains the object that was not found.

Object name used. The actual name of the object that was referenced. This name must always be filled in.

Object name specified. The name of the object referenced that was not found.

Object type. The type of object that was not found.

Record length. The length of this record is 56.

Record type. The type of this record is '15'.

Reserved. An ignored field.

Object already exists error record

This record is passed when the compiler or preprocessor fails because the object that was to be created exists. There is no REPLACE parameter on the command because the compiler or preprocessor expects the object not to exist.

After passing this record, the compiler or preprocessor must also pass the Abnormal processor end record (see Abnormal processor end record).

The Object already exists error record has the following format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	Object name that already exists
18	12	CHAR(10)	Object library name
28	1C	CHAR(7)	Object type
35	23	CHAR(1)	Reserved

Processors for which this record type applies

CRTPF	CRTLF	CRTDSPF	CRTICFF
CRTPRTF	processor processing the user-defined types represented as members		

Field Descriptions

Object library name. The name of the library that contains the object that already exists. A specific library name, not *CURLIB or *LIBL must be passed.

Object name that already exists. The name of the object that already exists and could not be replaced.

Object type. The type of the object that already exists.

Record length. The length of this record is 36.

Record type. The type of this record is '16'.

Reserved. An ignored field.

Start of new program record

The COBOL/400 compiler is able to compile source that contains more than one program. This record is passed by the COBOL/400 compiler when the beginning of a new program is encountered.

The Start of new program record has the following format:

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Record length
4	4	CHAR(2)	Record type
6	6	CHAR(2)	Reserved
8	8	CHAR(10)	New program name
18	12	CHAR(10)	Object name created
28	1C	CHAR(10)	Object library name
38	26	CHAR(7)	Message identifier
45	2D	CHAR(3)	Reserved
48	30	CHAR(7)	Object type
55	37	CHAR(1)	Reserved

Processors for which this record type applies

CRTCBLPGM	CRTCBLMOD	CRTBNDCBL	CRTSQLCBLI

processor processing the user-defined types represented as members

Field Descriptions

Message identifier. The message ID of the completion message.

New program name. The name of the new program, per IDENTIFICATION DIVISION.

Object library name. The library where the object was created. This field contains blank if an error occurred.

Object name created. The name of the object created in the previous step. If an object was not created because of syntax errors or because REPLACE(*NO) was specified and the object already existed, this field contains '*ERROR'.

Object type. The type of object created. For example, *PGM or *MODULE.

Record length. The length of this record is 56.

Record type. The type of this record is '40'.

Reserved. An ignored field.

Examples of Records Written

The following examples illustrate how compilers and preprocessors communicate with the Application Development Manager/400 APIs in different circumstances. In all these examples, assume that the compiles are submitted by an Application Development Manager/400 BLDPART command, which means it has called QLYSETS to set the status of the space to *READY before calling the compiler or preprocessor.

It is also assumed that a cleanup is done after the compile by calling QLYSETS again to set the status of the space to *COMPLETE.

Example 1

RPG/400 compiler successfully compiles source that has one include in it.

The compiler first calls QLYGETS and determines that it was started by the BLDPART command. Then it calls QLYWRTBI to pass records of the following record types and in the following order:

1. **Processor member start**
2. **Include**
3. **Normal processor end**

Example 2

DDS compiler successfully compiles source of type LF and creates a logical file based on two physical files.

The compiler first calls QLYGETS and determines that it was started by the BLDPART command. Then it calls QLYWRTBI to pass records of the following record types and in the following order:

1. **Processor member start**
2. **File reference**

   This record is called for the first physical file on which the logical file is based. The based-on indicator is set to Y (yes).
3. **File reference**

   This record is called for the second physical file on which the logical file is based. The based-on indicator is set to Y (yes).
4. **Normal processor end**

Example 3

COBOL/400 compiler fails when compiling source that has one include in it because the include was not found in *LIBL.

The compiler first calls QLYGETS and determines that it was started by a BLDPART command. Then it calls QLYWRTBI to pass records of the following record types and in the following order:

1. **Processor member start**
2. **Abnormal processor end**

Example 4

COBOL/400 compiler fails when compiling source that references a record format of a database file because the file was not found in *LIBL.

The compiler first calls QLYGETS and determines that it was started by a BLDPART command. Then it calls QLYWRTBI to pass records of the following record types and in the following order:

1. **Processor member start**
2. **External reference error**

   The name of the *Library specified* passed to QLYWRTBI is *LIBL.
3. **Abnormal processor end**

Example 5

ILE C CRTBNDC compiler successfully compiles a *PGM from a source that has one include in it.

The compiler calls QLYGETS and determines that it was started by the BLDPART command. Then it calls QLYWRTBI to pass records of the following record types and in the following order:

1. **Processor member start**
2. **Include**
3. **Normal processor end call next**
4. **Processor object start**
5. **Normal processor end**

Note: The Processor object start and the Normal processor end records are written by the CRTPGM processor internally called by the CRTBNDC compiler.

Example 6

CRTPGM binder successfully binds objects from 2 modules, and references a bind directory and a service program.

The compiler calls QLYGETS and determines that it was started by the BLDPART command. Then it calls QLYWRTBI to pass records of the following record types and in the following order:

1. **Processor object start**
2. **Module reference**
3. **Module reference**
4. **Bind directory reference**
5. **Service program reference**
6. **Normal processor end**.

Get Space Status (QLYGETS) API

Required Parameter Group:

1	Status	Output	Char(10)
2	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Get Space Status (QLYGETS) API obtains the status of the space.

Authorities and Locks

None

Required Parameter Group

Status

OUTPUT; CHAR(10)

**READY*	Information in the space is ready to be processed.
**COMPLETE*	Information in the space has been processed.
**NONE*	The space does not exist. Use QLYSETS to create the space.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Error Messages

Message ID	Error Message Text
CPF3CF1 E	Error code parameter not valid.
CPF3C90 E	Literal value cannot be changed.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

API Introduced: V2R2

Read Build Information (QLYRDBI) API

Required Parameter Group:

1	Buffer	Output	Char(*)
2	Maximum size	Input	Binary(4)
3	Read mode	Input	Char(10)
4	Buffer length	Output	Binary(4)
5	Number of records	Output	Binary(4)
6	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Read Build Information (QLYRDBI) API reads one or more records from the space.

QLYRDBI reads the space starting at the first location after the last record was read. If this is the first time QLYRDBI is called, the first record following the header record is read.

After QLYRDBI has read the final record, the next call to QLYRDBI starts reading the space from the beginning again.

QLYRDBI reads one or more records depending on the value specified on the Read mode parameter. QLYRDBI does not read more records than can fit in the buffer. The buffer is determined by the Maximum-size parameter.

Authorities and Locks

None.

Required Parameter Group

Buffer

OUTPUT; CHAR(*)

A character string to contain one or more records of build information.

Maximum size

INPUT; BINARY(4)

The maximum size of the data that is expected to be returned to this call. Maximum size should be large enough to fit at least one record. If it is too small for one record, an error occurs.

Read mode

INPUT; CHAR(10)

The mode of reading.

The possible read mode values are:

**SINGLE*	Read only one record.
**MULTIPLE*	Read more than one record. The maximum number of records that are read is determined by the size of Maximum size.

Buffer length

OUTPUT; BINARY(4)

The length of the data returned. If records are not read, 0 is returned.

Number of records

OUTPUT; BINARY(4)

The number of records read. Number of records is 0 if no records were read, 1 if one record was read or greater than 1 if *MULTIPLE was specified on read mode and more than one record could fit in the buffer.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Error Messages

The LIBxxxx error messages are located in the message file QLIBMSG in the QSYS library.

Message ID	Error Message Text
LIB9005	Value specified for Maximum size parameter is not valid.
LIB9006	Value specified for Read mode parameter is not valid.
LIB9007	Value specified for Maximum size parameter is too small.
LIB9009	Build information space does not exist, or it is damaged or deleted.
LIB9010	Build information missing or no more build information.
LIB9011	Build information in the space is not complete.
CPF3CF1 E	Error code parameter not valid.
CPF3C90 E	Literal value cannot be changed.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

API Introduced: V2R2

Top | High-level language APIs | APIs by category

Set Space Status (QLYSETS) API

Required Parameter Group:

1	Status	Input	Char(10)
2	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Set Space Status (QLYSETS) API sets the status of the space.

When QLYSETS is first called to create the space (if the space does not exist already) or to initialize the space so the information can be written to it by compilers or preprocessors, the Status parameter should be set to *READY. Then QLYSETS writes a special record (called the HEADER record) at the beginning of the space and initializes a status flag in that record to *READY. Now the space is ready to accept records containing build information. Compilers write to the space using the QLYWRTBI API. QLYWRTBI writes records to the space concatenated to each other. QLYRDBI later reads them sequentially in the order in which they are written.

Use the QLYSETS API to set the status flag in the space to *COMPLETE after the information in the space is processed using the QLYRDBI API. This indicates that the information in the space has been processed and the space can be reused.

Authorities and Locks

None

Required Parameter Group

Status

INPUT; CHAR(10)

The status for the space.

The possible status values are:

**READY*	Initialize the space. If the space does not exist, it is created.
**COMPLETE*	Information in the space has been processed. The space can now be used by setting it to *READY with another call to QLYSETS.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Error Messages

Message ID	Error Message Text
LIB9001	Value specified on the Status parameter is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3C90 E	Literal value cannot be changed.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

API Introduced: V2R2

Write Build Information (QLYWRTBI) API

Required Parameter Group:

1	Buffer	Input	Char(*)
2	Buffer length	Input	Binary(4)
3	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Write Build Information (QLYWRTBI) API writes one or more records to the space.

QLYWRTBI writes records to the space concatenated to each other. QLYRDBI later reads them sequentially in the order in which they are written.

QLYWRTBI continues to write records to the API space concatenated to previous records written, until QLYSETS is called. See Record Types for the records that can be written. See Examples of Records Written for examples of the sequence of records written.

Authorities and Locks

None.

Required Parameter Group

Buffer

INPUT; CHAR(*)

A character string containing one or more records of build information.

Buffer length

INPUT; BINARY(4)

The length of the buffer in bytes. The buffer length must be equal to the sum of the lengths of all the concatenated records being passed, otherwise an error occurs.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

The first field in each record indicates the record length. This allows all the records to be read sequentially using the QLYRDBI API.

Error Messages

Message ID	Error Message Text
LIB9002	Value specified for the buffer length parameter is not valid.
LIB9003	Value specified for the buffer length parameter is too small.
LIB9004	Record not in correct sequence.
LIB9008	Record has a record type that is not valid.
LIB9009	Build information space does not exist, or it is damaged or deleted.
CPF3CF1 E	Error code parameter not valid.
CPF3C90 E	Literal value cannot be changed.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

API Introduced: V2R2

COBOL/400 APIs

The OPM and ILE COBOL/400 APIs let you control run units and error handling.

Refer to Using COBOL Program to Call APIs and Error Handler for Example COBOL Program in the API Examples for illustrations of how to use these APIs.

For a description of how to use the ILE COBOL/400 APIs, refer to the chapter about error and exception handling in the WebSphere Development Studio: ILE COBOL Programmer's Guide book.

The COBOL/400 APIs are:

- Change COBOL Main Program (QLRCHGCM) lets you create an additional run unit (1) by assigning a different System/36-compatible COBOL, System/38-compatible COBOL, or iSeries OPM COBOL/400 program to serve as a main program.
- Dump COBOL (QlnDumpCobol) allows you to perform a formatted dump of an ILE COBOL/400 program.
- Retrieve COBOL Error Handler (QlnRtvCobolErrorHandler) allows you to retrieve the procedure pointer of the current COBOL error-handling procedure.
- Retrieve COBOL Error Handler (QLRRTVCE) allows you to retrieve the name of the current or pending COBOL error-handling program.
- Set COBOL Error Handler (QlnSetCobolErrorHandler) allows you to specify the identity of a COBOL error-handling procedure.
- Set COBOL Error Handler (QLRSETCE) allows you to specify the identity of a COBOL error-handling program.

The COBOL/400 exit programs are:

- ILE COBOL Error-Handling exit procedure acts as an error handler for an ILE COBOL/400 program.
- OPM COBOL Error-Handling exit program acts as an error handler for an OPM COBOL program.

Change COBOL Main Program (QLRCHGCM) API

Required Parameter

1	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Change COBOL Main Program (QLRCHGCM) API allows you to create an additional run unit by assigning a different System/36-compatible COBOL, System/38-compatible COBOL, or iSeries OPM COBOL/400 program to serve as a main program. You can call it from any programming language.

Note: By creating more than one run unit, you cantreat files, storage, and error conditions differently than you would using an ordinary subprogram.

After you call this API, the next nonactive COBOL program that runs becomes the main program in a new run unit. An active COBOL program is a program that has been called, and is not in its initial state.

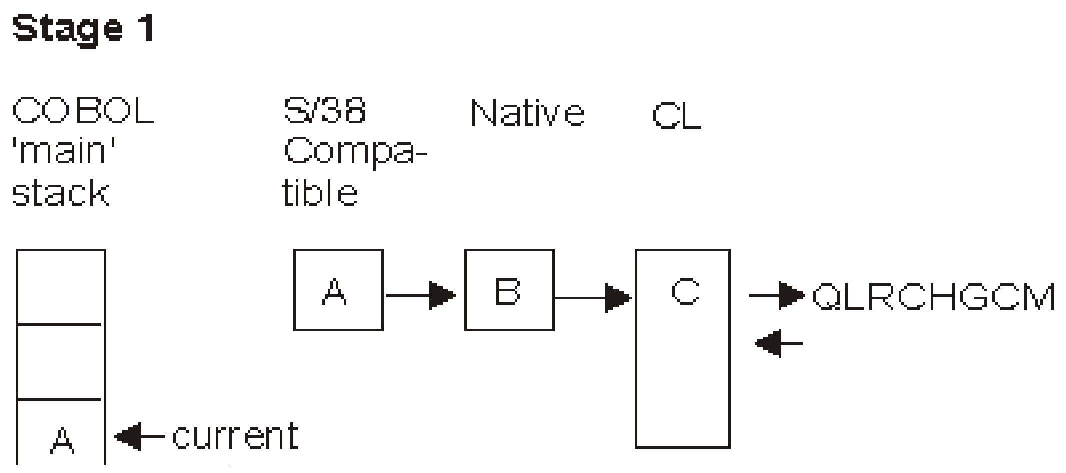
In the following example, System/38-compatible COBOL Program A calls iSeries COBOL/400 Program B. Because Program A is the first COBOL program, it is the main COBOL program.

COBOL Program B is a menu program that calls CL Program C.

Program C must start a new COBOL application that will pass control back to it, regardless of error conditions. To accomplish this, Program C calls the QLRCHGCM API before calling the new COBOL application.

When program C calls the new COBOL application in the form of Program D, Program D becomes the main program in a new run unit. When Program D's run unit ends, control returns to the original run unit, and Program A becomes the current main program again.

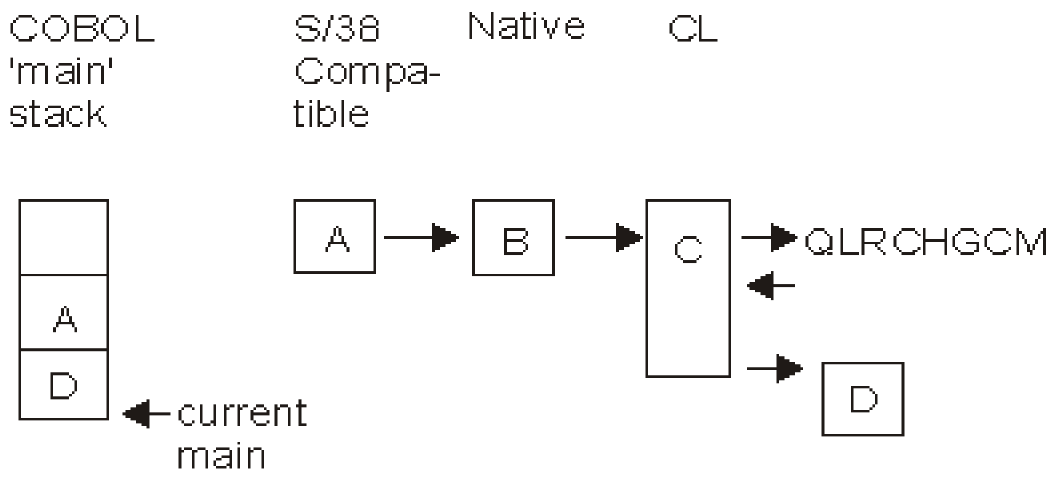
If, at the time a run unit is created, a program is active as a subprogram in an existing run unit, and this program is then called within the new run unit, it will be made available in its last-used state.



main

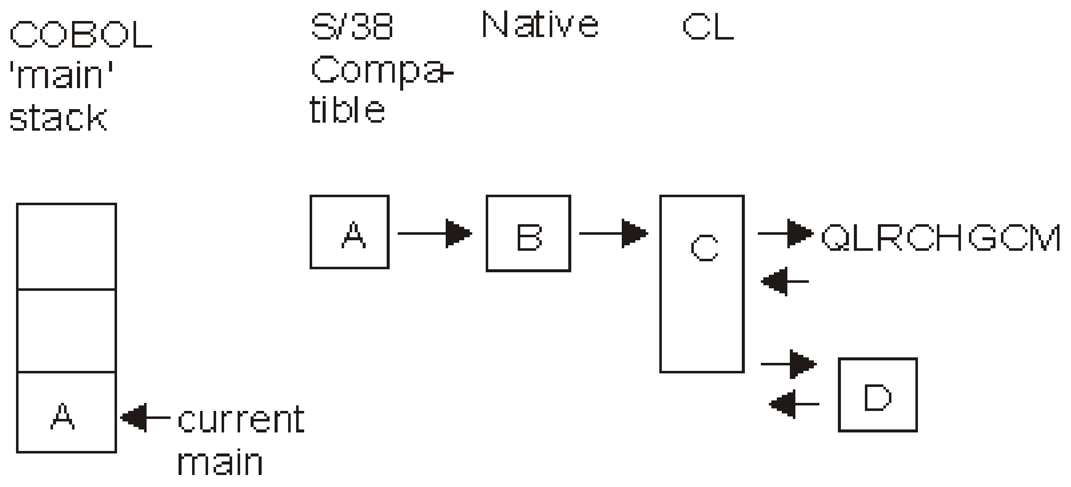
COBOL pending
main flag = ON

Stage 2



COBOL pending
main flag = OFF

Stage 3



COBOL pending

Required Parameter

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Error Messages

Message ID	Error Message Text
CPF3C90 E	Literal value cannot be changed.
LBE7040 E	Format of error code parameter is not correct.

API Introduced: V2R2

Dump COBOL (QlnDumpCobol) API

Required Parameter Group:

1	Program object name	Input	Char(10)
2	Library name	Input	Char(10)
3	Module object name	Input	Char(10)
4	Program object type	Input	Char(10)
5	Dump type	Input	Char(1)
6	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QLNRMAIN

Threadsafe: No

The Dump COBOL (QlnDumpCobol) API allows you to perform a formatted dump of an ILE COBOL/400 program. You can call it from any ILE program; however, if the calling program is not an ILE COBOL/400 program, only a data dump will be performed. Message CPF955F will be issued if this API is called to dump any module other than those created by the ILE COBOL/400 compiler.

This API provides two types of dumps, a data dump and an extended dump. The data dump contains the following information:

- The name of each variable
- The data type
- The default value
- The hexadecimal value

Note: Only the first 250 characters of the values will be shown in the dump.

The extended dump contains the following additional information:

- The name of each file
- The system name of each file
- External/internal flag
- Open/close status
- Last I/O operation attempted
- Last file status
- Last extended status
- Blocking information
- Blocking factor
- Linage-counter value
- I/O feedback area information
- Open feedback area information

Variable values may only be requested if an active call stack entry exists for the module object specified in the job in which this API is called. Values existing in program static or automatic storage are not accessible by this API unless the program object has a current call stack entry. All variables that were defined by the compiler and stored in the module object's HLL symbol table will be returned.

Also, the module object for which variable information is requested must contain debug data. Thus, the module object must be compiled with a *DBGVIEW option other than *NONE.

Required Parameter Group

Program object name

INPUT; CHAR(10)

The name of the program to be dumped. If this parameter is omitted, the program object name of the caller is used.

Library name

INPUT; CHAR(10)

The name of the library in which the program to be dumped is found. *CURLIB and *LIBL can be specified as valid values to indicate the current library and the library list, respectively. If this parameter is omitted, the library associated with the calling program is used.

Module object name

INPUT; CHAR(10)

The name of the module, within the specified program, to be dumped. If this parameter is omitted, the module object name of the caller is used.

Program object type

INPUT; CHAR(10)

The object type of the program object.

Valid values are:

**PGM*	Program object
**SRVPGM*	Service program

Dump type

INPUT; CHAR(1)

The type of dump.

Valid values are:

D	Data dump. Gives a dump of the COBOL identifiers.
F	Extended dump. Gives a dump of COBOL identifiers and file-related information.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code

Parameter.

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.
CPF3C90 E	Literal value cannot be changed.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9549 E	Error addressing API parameter.
CPF954F E	Module &1 not found.
CPF955F E	Program &1 not a bound program.
CPF9562 E	Module &1 cannot be debugged.
CPF956D E	Parameter does not match on continuation request.
CPF956E E	Program language of module not supported.
CPF956F E	Continuation handle parameter not valid.
CPF9573 E	Program type parameter not valid.
CPF9574 E	Call stack entry does not exist.
CPF9579 E	Data option specified not valid.
CPF9801 E	Object &2 in library &3 not found.
CPF9802 E	Not authorized to object &2 in &3.
CPF9803 E	Cannot allocate object &2 in library &3.
CPF9809 E	Library &1 cannot be accessed.
CPF9810 E	Library &1 not found.
CPF9820 E	Not authorized to use library &1.

API Introduced: V3R6

Retrieve COBOL Error Handler (QlnRtvCobolErrorHandler) API

Required Parameter Group:

1	Current error-handling exit procedure pointer	Output	Anyptr
2	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QLNRMAIN

Threadsafe: No

The Retrieve COBOL Error Handler (QlnRtvCobolErrorHandler) API allows you to retrieve the procedure pointer of the current COBOL error-handling procedure. You can call it from any ILE programming language; however, this API only retrieves the procedure pointer of the error handling program that is called when an error occurs in an ILE COBOL/400 program.

Required Parameter Group

Current error-handling exit procedure pointer

OUTPUT; ANYPTR

Valid values are:

NULL	No current error-handling procedure found.
procedure-pointer	The procedure pointer of the error handler.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error code parameter.

Error Messages

Message ID	Error Message Text
CPF3C90 E	Literal value cannot be changed.
LNR7074 E	Error code not valid.

LNR7075 E	Error addressing API parameters.

API Introduced: V2R1.1

Retrieve COBOL Error Handler (QLRRTVCE) API

Required Parameter Group:

1	Current or pending error-handling exit program name	Output	Char(20)
2	Scope of error-handling exit program	Input	Char(1)
3	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Retrieve COBOL Error Handler (QLRRTVCE) API allows you to retrieve the name of the current or pending COBOL error-handling program. You can call it from any programming language; however, this API only retrieves the name of the error handling program that is called when an error occurs in an OPM COBOL/400 program.

Required Parameter Group

Current or pending error-handling exit program name

OUTPUT; CHAR(20)

The qualified name of the error-handling program for the current or pending COBOL run unit.

The 20 characters of this parameter are:

1-10 The name of the program object.
Valid values are:

**NONE*	No user-defined COBOL error handler has been set.
program-name	The name of the error-handling program.

11-20 The library where the program object existed.
The valid value is:

library-name	The library where the program object existed.

Scope of error-handling exit program

INPUT; CHAR(1)

The program can apply to a current or pending run unit.

Valid values are:

C	Current COBOL run unit
P	Pending COBOL run unit

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Error Messages

Message ID	Error Message Text
CPF3C90 E	Literal value cannot be changed.
LBE7040 E	Format of error code parameter is not correct.
LBE7051 E	Scope parameter not valid.
LBE7052 E	Run unit specified for error handler does not exist.
LBE7055 E	Severe error while addressing parameter list. The API did not complete.

API Introduced: V3R6

Set COBOL Error Handler (QlnSetCobolErrorHandler) API

Required Parameter Group:

1	New error-handling exit procedure pointer	Input	Anyptr
2	Current error-handling exit procedure pointer	Output	Anyptr
3	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Set COBOL Error Handler (QlnSetCobolErrorHandler) API allows you to specify the identity of a COBOL error-handling procedure. You can call it from any ILE programming language; however, this API only sets the procedure pointer of the error-handling program that is called when an error occurs in an ILE COBOL/400 program.

After you call this API, any ILE COBOL/400 program that issues an inquiry message with options C, D, or F will first call the defined error-handling procedure. This procedure receives the message identification and substitution text, as well as the name of the program that received it, and a list of valid 1-character responses. The defined procedure is responsible for returning a 1-character code (blank, C, D, F, or G) indicating whether the COBOL program should continue or not.

Note: All messages issued by the operating system during the running of a COBOL program are monitored by the COBOL program. Only some of the system messages issued will result in a COBOL inquiry message.

You can define a different error-handling procedure for each activation group.

Only one ILE error-handling procedure can be active at a time. If an error occurs in the error-handling procedure, the COBOL program does not call the error-handling procedure again. (In other words, recursive calls do not occur.) Instead, the inquiry message would be issued as if no error-handling procedure were defined.

You cannot change the error-handling procedure while it is responding to an error in a COBOL program.

If an error occurs during the calling of the error-handling procedure, an informational message (LNR7430) is issued, and processing continues as if no error-handling procedure were defined.

The error-handling procedure is defined by the user. The parameters aredescribed under ILE COBOL Error-Handling Exit Procedure.

Required Parameter Group

New error-handling exit procedure pointer

INPUT; ANYPTR

The pointer to the new error-handling procedure that you want to set.

Current error-handling exit procedure pointer

OUTPUT; ANYPTR

The pointer to the error-handling procedure that was in place before the new error-handling procedure was set.

Valid values are:

NULL	No current error-handling exit procedure was found.
procedure-pointer	The pointer to the error-handling procedure.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error code parameter.

Error Messages

Message ID	Error Message Text
CPF3C90 E	Literal value cannot be changed.
LNR7074 E	Error code not valid.
LNR7075 E	Error addressing API parameters.
LNR7077 E	Procedure reference not valid.

API Introduced: V2R2

Set COBOL Error Handler (QLRSETCE) API

Required Parameter Group:

1	Error-handling exit program name	Input	Char(20)
2	Scope of error-handling program	Input	Char(1)
3	New error-handling exit program library	Output	Char(10)
4	Current or pending error-handling exit program name	Output	Char(20)
5	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Set COBOL Error Handler (QLRSETCE) API allows you to specify the identity of a COBOL error-handling program. You can call it from any programming language; however, this API only sets the name of the error handling program that is called when an error occurs in an OPM COBOL/400 program.

After you call this API, any COBOL/400 program that issues an inquiry message with options C, D, or F will first call the defined error-handling program. This program receives the message identification and substitution text, as well as the name of the program that received it, and a list of valid one-character responses. The defined program is responsible for returning a one-character code (blank, C, D, F, or G) indicating whether the COBOL program should continue or not.

Note: All messages issued by the operating system during the running of a COBOL program are monitored by the COBOL program. Only some of the system messages issued will result in a COBOL inquiry message.

For more information about error handling and the issuing of COBOL inquiry messages, see the chapter on error handling in the WebSphere Development Studio: ILE COBOL Programmer's Guide book.

You can define a different error-handling program for each COBOL run unit, but when a new COBOL run unit starts, it uses the error-handling program from the previous run unit.

Only one error-handling program can be active at a time. If an error occurs in the error-handling program, the COBOL program does not call the error-handling program again. (In other words, recursive calls do not occur.) Instead, the inquiry message would be issued as if no error-handling program were defined.

You cannot change the name of the error-handling program while it is responding to an error in a COBOL program.

If an error occurs during the calling of the error-handling program, an informational message (LBE7430) is issued, and processing continues as if no error-handling program were defined.

The error-handling program is defined by the user. The parameters are described under OPM COBOL Error-Handling Exit Program.

Required Parameter Group

Error-handling exit program name

INPUT; CHAR(20)

The qualified name of the error-handling program.

The 20 characters of this parameter are:

1-10 The name of the program object.
Valid values are:

**NONE*	No user-defined COBOL error-handling program exists.
program-name	The name of the error-handling program. The name can be an extended one.

11-20 The library where the program object exists.
Valid values are:

**CURLIB*	The current library is used.
**LIBL*	The API searches the library list to find the object.
library-name	The name of the library where the program object exists. The name can be an extended one.

Scope of error-handling program

INPUT; CHAR(1)

The program can apply to a current or pending run unit.

Valid values are:

C	Current COBOL run unit
P	Pending COBOL run unit

New error-handling exit program library

OUTPUT; CHAR(10)

The library where the program object exists. If *CURLIB or *LIBL was specified for the error-handling exit program name parameter, the library returned for this parameter shows the library where the program was found. If *CURLIB or *LIBL was not specified, the library returned here should be the same as character 11 through 20 of the error-handling exit program name parameter.

Valid value is:

library-name	The library where the program object exists.

Current or pending error-handling exit program name

OUTPUT; CHAR(20)

The qualified name of the error-handling program that was in place before the current error-handling program was set.

The 20 characters of this parameter are:

1-10	The name of the previous error-handling program object. Valid values are:
	**NONE* — No previous current or pending error-handling program existed.
	program-name — The name of the error-handling program.
11-20	The library where the previous error-handling program object existed. Valid value is:
	library-name — The library where the previous error-handling program object existed.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Error Messages

Message ID	Error Message Text
CPF3C90 E	Literal value cannot be changed.
LBE7040 E	Format of error code parameter is not correct.
LBE7050 E	Error handler is already responding to an error in the same run unit.
LBE7051 E	Scope parameter not valid.
LBE7052 E	Run unit specified for error handler does not exist.
LBE7055 E	Severe error while addressing parameter list.The API did not complete.
LBE7060 E	Error in program name or availability.
LBE7061 E	Error in library name or availability.
LBE7062 E	Error in library list.

API Introduced: V3R6

ILE COBOL Error-Handling Exit Procedure

Required Parameter Group:

1	COBOL message identification	Input	Char(7)
2	Valid responses to message	Input	Char(6)
3	Name of program issuing error	Input	Char(20)
4	System message causing COBOL message	Input	Char(7)
5	Length of passed message text	Input	Binary(4)
6	Return code	Output	Char(1)
7	Message text	Input	Char(*)
8	Module name	Input	Char(10)
9	COBOL program name	Input	Char(256)

This is a user-defined program that acts as an error handler for an ILE COBOL/400 program. Use the Set COBOL Error Handler (QlnSetCobolErrorHandler) API to establish this relationship between the two programs.

Required Parameter Group

COBOL message identification

INPUT; CHAR(7)

A 3-character prefix followed by a 4-character number.

Valid responses to message

INPUT; CHAR(6)

The list of valid 1-character responses. This list is variable in length and consists of uppercase letters in alphabetical order. The list always ends with a space.

The following are examples of lists of valid responses:

CG

CDFG

Name of program issuing error

INPUT; CHAR(20)

The qualified name of the ILE COBOL/400 program that issued the error.

The 20 characters of this parameter are:

1-10	The name of the program object. The valid value is: *program-name* The name of the program object.
11-20	The library where the program object existed. The valid value is: *library-name* The library where the program object existed.

System message causing COBOL message

INPUT; CHAR(7)

Some COBOL error messages are issued because of error messages received from the system. This parameter identifies such system messages.

Valid values are:

*NONE	No system message is available.
message-id	A 3-character message prefix followed by a 4-character number.

Length of passed message text

INPUT; BINARY(4)

If the original message was a system message, the substitution text for the system message is passed. In the absence of an original system message, Parameter 4 has a value of *NONE, and the substitution text for the COBOL message is passed.

Return code

OUTPUT; CHAR(1)

Must be one of the values specified in Parameter 2, or a space. If the value is not one of these, a response of a space is assumed.

Valid values are:

blank	Issue the COBOL message that was passed to the error-handling program.
G	Continue running the COBOL program.
C	End the current COBOL run unit.
D	Same as C, but produce a formatted dump of user-defined COBOL variables.
F	Same as D, but also dump COBOL's file-related internal variables.

Message text

INPUT; CHAR(*)

The substitution text of the message. Its length is determined by Parameter 5.

Module name

INPUT; CHAR(10)

The module within the program object that issued the error.

COBOL program name

INPUT; CHAR(256)

The name of the COBOL program, from the PROGRAM-ID paragraph, that issued the error.

Exit program introduced: V3R2

OPM COBOL Error-Handling Exit Program

Required Parameter Group:

1	COBOL message identification	Input	Char(7)
2	Valid responses to message	Input	Char(6)
3	Name of program issuing error	Input	Char(20)
4	System message causing COBOL message	Input	Char(7)
5	Message text	Input	Char(*)
6	Length of passed message text	Input	Binary(4)
7	Return code	Output	Char(1)

This is a user-defined program that acts as an error handler for an OPM COBOL program. Use the Set COBOL Error Handler (QLRSETCE) API to establish this relationship between the two programs.

Required Parameter Group

COBOL message identification

INPUT; CHAR(7)

A 3-character prefix followed by a 4-character number.

Valid responses to message

INPUT; CHAR(6)

The list of valid 1-character responses. The list is variable in length and consists of uppercase letters in alphabetical order. The list always ends with a space.

Examples of lists of valid responses:

CG

CDFG

Name of program issuing error

INPUT; CHAR(20)

The qualified name of the COBOL/400 program that issued the error.

The 20 characters of this parameter are:

1-10	The name of the program object. The valid value is:

program-name	The name of the program object.

The library where the program object existed.
The valid value is:

library-name	The library where the program object existed.

System message causing COBOL message

INPUT; CHAR(7)

Some COBOL error messages are issued because of error messages received from the system. This parameter identifies such system messages.

Valid values are:

**NONE*	No system message is available.
message-id	A 3-character message prefix followed by a 4-character number.

Message text

INPUT; CHAR(*)

The substitution text of the message, its length determined by Parameter 6.

Length of passed message text

INPUT; Binary(31)

If the original message was a system message, the substitution text for the system message is passed. In the absence of an original system message, Parameter 4 has a value of *NONE, and the substitution text for the COBOL message is passed.

Return code

OUTPUT; CHAR(1)

Must be one of the values specified in Parameter 2, or a space. If the value is not one of these, a response of a space is assumed.

Valid values are:

blank	Issue the COBOL message that was passed to the error-handling program.
G	Continue running the COBOL program.
C	End the current COBOL run unit.
D	Same as C, but produce a formatted dump of user-defined COBOL variables.
F	Same as D, but also dump COBOL's file-related internal variables.

Exit Program Introduced: V3R2
