Debugger APIs (V5R2)

Table of Contents

Debugger APIs

The Debugger APIs can be used for program debugging on the server. The Debugger APIs include:

- Source Debugger APIs (Integrated Language Environment (ILE) APIs)
- Retrieve Program Variable API (Original program model (OPM) API)

The debugger API user can use these APIs independently of each other or together as needed. For general information about the integrated language environment, see the ILE Concepts book.

Select one of the following for more information:

- Using source debugger APIs
- How a compiler uses the APIs to generate debug data for ILE programs
- How a source debugger uses the APIs to debug ILE or OPM programs

---

Using Source Debugger APIs

The Source Debugger APIs can be used to write debuggers for the iSeries. The users of these APIs include:

- The source debugger that is shipped with the OS/400 licensed program. A **source debugger** is a tool for debugging Integrated Language Environment (ILE) programs or original program model (OPM) programs by displaying a representation of their source code.
- Any other debugger that IBM or a business partner writes.

Debugger functions are designed to help you write and maintain your applications. You can run your programs in a special testing environment while closely observing and controlling the processing of these programs in the testing environment. You can write a debugger application that interacts with the APIs provided in this chapter, or you can use the debugger provided with the system.

No special commands specifically for testing are contained in the program being tested. The same program being tested can run normally without changes. All debugger APIs must be called within the job in which the Start Debug (STRDBG) command is issued. The debugger APIs should not be called from within the program being tested. With the debugger APIs provided, you interact with your programs symbolically in the same terms as the high-level language (HLL) in the program. You refer to variables by their names and to locations as the line and the column within a view. In addition, the debugger functions are only applicable to the job in which they are set up. The same program can be used at the same time in another job without being affected by the debugger functions set up.

How a compiler uses the APIs to generate debug data for ILE programs

To enable source-level debugging of ILE programs, view information must be stored with the compiled program. The ILE compilers use the Create View APIs to create view information. This information is then available to source-level debugger applications through the Source Debugger APIs.

The first API that is called is the Start View Creation (QteStartViewCreation) API, which is used to initialize the debug view creation environment.

The views being created are described by the Add View Description (QteAddViewDescription) API. Examples of views created by a compiler are text views (for example, the input source) and listing views (for example, a compiler output listing). A parameter passed back by this API is the view number, which is used by subsequent APIs to identify the view being processed.

The text of a view comes from files (for example, the input source file to the compiler) or supplied text (for example, macro expansion text). The supplied text is stored with the view information in the program object. The file text is copied at source debugging time when the text is retrieved. Thus, the view information stored for the file text contains references to the files containing the text and not the text itself. The files to be used in a view are described by use of the Add View File (QteAddViewFile) API.

The Add View Text (QteAddViewText) API is used to describe how the text for a view is constructed. The view text can be composed of pieces of text, which are concatenated together when the text is retrieved, according to the instructions specified through this API.

The Add View Map (QteAddViewMap) API is used to map positions in one view to positions in another view. This is necessary to be able to relate positions in one view to equivalent positions in another view. In some cases a map can be generated automatically without using this API (see QteAddViewDescription API). Other maps may need to be supplied to allow certain source debugger functions such as breakpoint processing, in which the breakpoint parameters are supplied by the system in terms of the statement view only.

When the view creation processing is complete, a call to the End View Creation (QteEndViewCreation) API is required.

To use a Create View API, the application must bind to the service program QTECRTVS in QSYS. All Create View API functions are then available to the application.

How a source debugger uses the APIs to debug ILE or OPM programs

The Start Debug command has a parameter, SRCDBGPGM, that specifies which program is called when an ILE or OPM program is debugged. The system calls this program, indicating that the debug session is to begin. It also calls this program when the user wants to show the Display Module Source display. When OPM programs are to be debugged, the additional OPMSRC(*YES) parameter must be specified on the Start Debug command.

When the system calls the source debugger program, indicating the start of a debug session, that program uses source debugger APIs to perform debug functions. The first API that is called is the Start Source Debug (QteStartSourceDebug) API, which indicates to the system that a source debugger is running.

When an ILE program is debugged, the Retrieve Module Views (QteRetrieveModuleViews) API is used to obtain information about the views available in the modules of that program. For an OPM program, information about the views available for that program is obtained. These views previously were created by the compiler by using the create view APIs for ILE programs. For OPM programs, the views were created by using OPTION(*SRCDBG) or OPTION(*LSTDBG) on the appropriate OPM language create program command. The OPM CL, COBOL, and RPG languages are supported by the source debugger APIs. A **view** is text that is displayed by the source debugger. A module may have several views, depending on the debug data supplied by the compiler of that module. OPM programs always have a statement view, and either a source or listing view, depending on the OPM compiler option selected. See the appropriate language reference manual to determine which views are available.

To be debugged, a module has to have at least one view: the statement view. A **statement view** is a low-level view that contains information about each high-level statement in that module. This view is not meant to be displayed, although there is text associated with that view. The information in the statement view text can be used by the source debugger to determine the following:

- Procedure name
- Statement number
- Statement type associated with any high-level language statement in the module

The source debugger application uses the Register Debug View (QteRegisterDebugView) API to register the views of a program. Once these views are registered, various debug operations can be performed against these views. These operations include:

- Retrieving the text associated with the views
- Adding a breakpoint to the program at a certain location in a view
- Displaying variables that are defined in the program

The source debugger application uses the Retrieve View Text (QteRetrieveViewText) API to retrieve the text of a view. Every view has text associated with it that can be retrieved using the QteRetrieveViewText API.

When a program is being debugged and it stops at a breakpoint, the system indicates that it has stopped by calling the Program-Stop Handler exit program. This program is passed a line number in the statement view where the program being debugged has stopped.

The Map View Position (QteMapViewPosition) API is used to map positions in one view to positions in another view. For example, if the source debugger currently is displaying a source view in a module, and a breakpoint occurs, the Program-Stop Handler exit program is called. This program is passed a line number in the statement view of that module, which indicates at which statement the program has stopped. To show the position in the source view where the program has stopped, the application maps the statement view position to a source view position. This mapping function is made possible by maps, which are created by the ILE compiler using the create view APIs, or by the debug data, which is created by OPM compilers.

When the debug session is over, the source debugger application issues the End Source Debug (QteEndSourceDebug) API, which removes all ILE and OPM programs from debug mode. No source debugger APIs can be issued until the source debug session is ended with the End Debug Command and started again with the Start Debug command.

The Create View APIs require the application to bind to the service program QTECRTVS in library QSYS. All other Source Debugger APIs require the application to bind to the service program QTEDBGS in library QSYS. All source debugger API functions are then available to the application.

For a coding example of how to write a source debugger, see Using Source Debugger APIs in the API examples.

Source Debugger APIs

The source debugger APIs are divided into the following functional areas:

- Debug Session Control APIs
- Create View APIs
- View Information APIs
- Fast-path Debugger APIs
- Submit Debug Command API, which allows a program to issue debug language statements. Debug language statements permit programs to enter breakpoints, run one or more statements of a program being debugged, and evaluate expressions. Debug commands are a part of the API that takes on free-form expressions. They are referred to as the debug language that the the program may supply to the source debugger support.
- Dump Module Variables API, which gets a list of all the variable names and current values of those variables.

Debug Session Control APIs

Debug session control APIs are used to start the source debug session, determine which programs, modules, and views are referenced, and control certain attributes of the debug environment.

The Debug Session Control APIs are:

- Change Current Thread (QteChangeCurrentThread) changes the current thread to any thread being debugged.
- Change Thread Status (QteChangeThreadStatus) changes the debug status for threads being debugged.
- End Source Debug (QteEndSourceDebug) takes the job out of debug mode.
- »Register Service Entry Point Stop Handler (QteRegSrvEntPntStpHdlr) registers a special event handler to handle Service Entry breakpoint events.«
- Retrieve Debug Attribute (QteRetrieveDebugAttribute) retrieves the attributes of the source debug session.
- Retrieve Debugged Threads (QteRetrieveDebuggedThreads) retrieves information for threads being debugged.
- Retrieve Module Views (QteRetrieveModuleViews()) returns to the caller the list of modules and views that are associated with a specific ILE or OPM program.
- Retrieve Source Path Name (QteRetrieveSourcePathName) returns the full source path name for a Java source file.
- Set Debug Attribute (QteSetDebugAttribute) sets the attributes of the source debug session.
- Start Source Debug (QteStartSourceDebug()) enables your session to use the source debugger.
- Stop Debugged Job (QteStopDebuggedJob) causes debug to halt all threads being debugged in a job.

Debug session control exit programs are used to process breakpoints and start, stop, and display screens within the source debug session. The debug session control exit programs are:

- Debug Session Handler manages the source debugger, telling it when to start, stop, and display its screens. This program is registered on the Start Debug Command.
- Program Stop Handler is registered on the Start Source Debug API. This program is called by the source debugger support when an ILE or OPM program stops at a breakpoint or for other reasons.
- »Service Entry Point Stop Handler is a user-written program that handles the service entry point program-stop condition.«

Change Current Thread (QteChangeCurrentThread) API

Required Parameter Group:

1	Thread ID	Input	Char(8)
2	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTETHRD

Threadsafe: No

The Change Current Thread (QteChangeCurrentThread) API changes the current thread to any thread being debugged.

> **Note:** A job may have several threads, each of which are debugged if the job is debugged. By default, the current thread is the initial thread of the debugged job, or the thread at a debug stop.

A current thread has several properties that distinguish it from other threads being debugged:

- Debug commands are run in this thread.
- This is the first thread released after a return to the base debug support after a debug stop (breakpoint, step, watch, or unmonitored exception).
- If there are multiple debug stops at the same time, this thread is the first one processed.

Before the current thread can be changed the new current thread must be stopped or halted and is waiting for debug to restart it. If this is not true, an error is returned to the API caller and the current thread is not changed.

Threads debugging is supported if a service job is used to debug a job that was spawned by native threads support. If this is not the debug environment present when this API is called, a CPF958B error is returned.

Required Parameter Group

Thread ID

INPUT; CHAR(8)

This is an 8-byte handle assigned by the system.

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.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9549 E	Error addressing API parameter.
CPF9589 E	Thread &1 not stopped or halted.
CPF958A E	Thread &1 not found.
CPF958B E	Threads debugging not supported.

API Introduced: V4R2

Change Thread Status (QteChangeThreadStatus) API

Required Parameter Group:

1	Thread debug status	Input	Char(10)
2	Thread array	Input	Array of Char(8)
3	Number of threads	Input	Binary(4)
4	Error code	I/O	Char(*)

Service Program: QTETHRD

Threadsafe: No

The Change Thread Status (QteChangeThreadStatus) API changes the debug status for threads being debugged.

> **Note:** A job may have several threads, each of which are debugged if the job is debugged. By default, the current thread is the initial thread of the debugged job, or the thread at a debug stop.

Before the debug status of a thread can be changed, the thread must be stopped or halted by debug support. If any thread specified in the thread array has not been stopped or halted by debug support, an error is returned to the API caller and the debug status of all threads is unchanged.

Threads debugging is supported if a service job is used to debug a job that was spawned by native threads support. If this is not the debug environment present when this API is called, a CPF958B error is returned.

Authorities and Locks

None

Required Parameter Group

Thread debug status

INPUT; CHAR(10)

The desired debug status for the thread identifiers specified in the thread array parameter. The valid debug status values are:

**ENABLE*	Enable the specified threads.
**DISABLE*	Disable the specified threads.

Thread array

INPUT; ARRAY OF CHAR(8)

The thread identifiers for which debug status is changed. Each thread identifier in the thread array is 8 bytes long. The number of thread identifiers is specified in the number of threads parameter.

If the number of threads parameter is minus one, the first and only thread array parameter must be a special value. In this case, all other thread array parameters are ignored. Valid special values are:

**ALL*	The debug status for all threads is changed to the debug status specified in the thread debug status parameter.

Number of threads

INPUT; BINARY(4)

The number of thread identifiers provided in the thread array parameter. The number of threads parameter must be greater than zero or a minus one. If it has a value of minus one, the first and only thread array parameter must be a special value. If it is greater than zero, the number specified is the number of thread array parameters that must be provided.

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
CPF3C1E E	Required parameter &1 omitted.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode
CPF9549 E	Error addressing API parameter.
CPF958A E	Thread &1 not found.
CPF958B E	Threads debugging not supported.
CPF958C E	Number of threads not valid.
CPF959B E	Thread status value not valid.
CPF959C E	Thread array special value not valid.
CPF959D E	Thread status cannot be changed.

API Introduced: V4R3

End Source Debug (QteEndSourceDebug) API

Required Parameter Group:

1	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTEDBGS

Threadsafe: No

The End Source Debug (QteEndSourceDebug) API is used to end the source debug support. All ILE and OPM programs being debugged under the source debug support are removed from debug mode. All registered views to programs being debugged are no longer valid.

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
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9549 E	Error addressing API parameter.

API Introduced: V2R3

»Register Service Entry Point Stop Handler (QteRegSrvEntPntStpHdlr) API

Required Parameter Group:

1	Qualified program name	Input	Char(20)
2	Error Code	I/O	Char(*)

Service Program Name: QTEDBGS

Default Public Authority: *USE

Threadsafe: No

The **Register Service Entry Point Stop Handler API** registers a special event handler to handle Service Entry breakpoint events. The Start Source Debug (QteStartSourceDebug) API must be called before this API can be issued.

Authorities and Locks

Program Authority
*EXECUTE

Library Authority
*EXECUTE

Required Parameter Group

Qualified program name
INPUT; CHAR(20)

The name of the exit program that is called when a service entry point is encountered.

The first 10 characters contain the program name. The second 10 characters contain the name of the library in which the program is located. Both entries must be left-justified.

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	Error code parameter not valid.
CPF3CF2	Errors occurred during running of API.
CPF9541	Not in debug mode.
CPF9549	Error addressing API parameter.
CPF9803	Cannot allocate object in library.
CPF9809	Library cannot be accessed.
CPF9810	Library not found.
CPF9811	Program in library not found.
CPF9820	Not authorized to use library.
CPF9821	Not authorized to program in library.

«

API introduced: V5R2

Retrieve Debug Attribute (QteRetrieveDebugAttribute) API

Required Parameter Group:

1	Debug attribute	Input	Char(10)
2	Attribute value	Output	Char(10)
3	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Retrieve Debug Attribute (QteRetrieveDebugAttribute) API is used to retrieve the attributes of the source debug session. These attributes may be any of the following:

- Default attributes established when the debug session was started
- Attributes changed with the Set Debug Attribute API
- Attributes changed by the Change Debug (CHGDBG) command

The attributes of the debug environment cannot be retrieved unless the job is currently in debug mode.

Required Parameter Group

Debug attribute

INPUT; CHAR(10)

The name of the debug environment attribute that is retrieved. The valid values for this parameter are:

**UPDPROD*	Retrieves the value of the update production files attribute.
**DEBUGJOB*	Retrieves an indicator of which job is being debugged.
**OPMSRC*	Retrieves the value of the OPM source debug attribute.

Attribute value

OUTPUT; CHAR(10)

The current value of the attribute identified in the debug attribute parameter.

When the debug attribute parameter contains *UPDPROD, the attribute value parameter can have one of the following values:

**YES*	Allow the updating of production files while in debug mode.

**NO*	Do not allow the updating of production files while in debug mode.

When the debug attribute parameter contains *DEBUGJOB, the attribute value parameter can have one of the following values:

**LOCAL*	The debug session is debugging programs that run in the job in which this API is running.
**REMOTE*	The debug session is debugging programs that run in the job specified in the Start Service Job (STRSRVJOB) command.

When the debug attribute parameter contains *OPMSRC, the attribute value parameter can have one of the following values:

**YES*	Allow OPM programs that have source debug data to be debugged by using the ILE debug APIs.
**NO*	Do not allow OPM programs to be debugged by using the ILE debug APIs.

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.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9549 E	Error addressing API parameter.
CPF9559 E	Debug attribute parameter not valid.

API Introduced: V2R3

Retrieve Debugged Threads (QteRetrieveDebuggedThreads) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	Format name	Input	Char(8)
4	Thread array	Input	Array of Char(8)
5	Number of threads	Input	Binary(4)
6	Error code	I/O	Char(*)

Service Program: QTETHRD

Threadsafe: No

The Retrieve Debugged Threads (QteRetrieveDebuggedThreads) API retrieves information for threads being debugged.

> **Note:** A job may have several threads, each of which is debugged if the job is debugged. By default, the current thread is the initial thread of the debugged job or the thread at a debug stop.

Information about the requested threads is returned in the receiver variable. This allows the writer of a debugger to maintain and control a list of threads that are being debugged. If this API is processed when threads are active, the information returned by the API may no longer be accurate. Check the job status flag to see what state the job was in when the API was processed.

Threads debugging is supported if a service job is used to debug a job that was spawned by native threads support. If this is not the debug environment present when this API is called, a CPF958B error is returned.

Authorities and Locks

None

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The receiver variable that receives the information requested. You can specify the size of this area to be smaller than the format requested as long as you specify the length parameter correctly. As a result, the API returns only the data the area can hold. For more information, see Format of Receiver Variable. Entries are only returned in their entirety. The API never returns anything less.

If there is not enough space for the entire entry, that entry is not returned and bytes available and bytes returned are not equal.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. The length of receiver variable parameter may be specified up to the size of the receiver variable specified in the user program. If the length of receiver variable parameter specified is larger than the allocated size of the receiver variable in the user program, the results are not predictable. The minimum length is 8 bytes.

Format name

INPUT; CHAR(8)

The content and format of the information returned in the receiver variable. The possible format names are:

THDL0100	Basic thread debug information.
THDL0200	Extended thread debug information.

Thread array

INPUT; ARRAY OF CHAR(8)

The thread identifiers (IDs) for which debug information is returned. In the thread array parameter, thread IDs are specified and debug information about the requested threads is returned in the receiver variable. Each thread identifier in the thread array is 8 bytes long. The number of thread identifiers is specified in the number of threads parameter.

If the number of threads parameter is minus one, the first thread array parameter must be a special value. In this case, all other thread array parameters are ignored. Valid special values are:

**ALL*	Thread debug information for all threads is returned.
**CURRENT*	Thread debug information for the current thread is returned.
**INITIAL*	Thread debug information for the initial thread is returned.
**ENABLE*	Thread debug information for all enabled threads is returned.
**DISABLE*	Thread debug information for all disabled threads is returned.

Number of threads

INPUT; BINARY(4)

The number of thread identifiers provided in the thread array parameter. The number of threads parameter must be greater than zero or minus one. If it has a value of minus one, the first and only thread array parameter must be a special value. If it is greater than zero, the number specified is the number of thread array parameters that must be provided.

Error code

I/O; CHAR(*)

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

Format of Receiver Variable

The following receiver variable formats are returned based on the format name parameter:

THDL0100 Format

The following table shows the format of the receiver variable for the THDL0100 format. For more information on the fields, see Field Descriptions.

Offset		Type	Field
Dec	**Hex**		
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	CHAR(1)	Job status flag
9	9	CHAR(3)	Reserved
12	C	BINARY(4)	Offset to thread records
16	10	BINARY(4)	Number of thread records
20	14	BINARY(4)	Size of thread record
		CHAR(*)	Reserved
Note: The following fields repeat the number of times specified in the number of thread records field.			
		CHAR(8)	Thread ID
		CHAR(1)	Current thread flag
		CHAR(1)	Initial thread flag
		CHAR(1)	Thread run state
		CHAR(1)	Thread debug status

THDL0200 Format

The following table shows the format of the receiver variable for the THDL0200 format. For more information on the fields, see Field Descriptions.

Offset		Type	Field
Dec	**Hex**		
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	CHAR(1)	Job status flag
9	9	CHAR(3)	Reserved
12	C	BINARY(4)	Offset to thread records
16	10	BINARY(4)	Number of thread records
20	14	BINARY(4)	Size of thread record
		CHAR(*)	Reserved

Note: The following fields repeat the number of times specified in the number of thread records field.			
		CHAR(8)	Thread ID
		CHAR(1)	Current thread flag
		CHAR(1)	Initial thread flag
		CHAR(1)	Thread run state
		CHAR(1)	Thread debug status
		CHAR(3)	Reserved
		CHAR(1)	top of stack flag
		BINARY(4)	Statement view ID stopped in
		BINARY(4)	Line in statement view stopped in

Field Descriptions

Bytes available. The number of bytes of data available to be returned to the user.

Bytes returned. The number of bytes of data returned to the user.

Current thread flag. Whether the thread is the current thread or not. Possible values are:

0	The thread is not the current thread.
1	The thread is the current thread.

Initial thread flag. Whether the thread is the initial thread or not. Possible values are:

0	The thread is not the initial thread.
1	The thread is the initial thread.

Job status flag. The status of the job when the API was processed.

0	The job is stopped by debug. The information returned by this API is accurate.
1	The job is running and has not been stopped by debug (for example, breakpoint, step, watch, or unmonitored exception). If threads are running it is not possible for debug to present a stable debugging environment. The information returned by this API may no longer be accurate.

Line in statement view stopped in. If the thread is stopped in a module that has been registered under debug, this is the line number in the module's statement view where the thread is stopped. See the statement view ID stopped in field for more information. This field is only applicable for the current thread. If the thread being returned is not the current thread then this field will contain a -1.

Number of thread records. The number of thread records that are returned in the receiver variable. Each record has the same format, and is repeated in the receiver variable.

Offset to thread records. The offset in bytes from the start of the receiver variable to the first requested thread information record.

Reserved. An ignored field.

Size of thread record. The number of bytes occupied by each thread record.

Statement view ID stopped in. The view ID of a previously registered debug statement view. It is the statement view ID of the highest module found on the call stack that has been registered under debug. If no statement views on the stack are registered, the thread is not stopped by debug, or if the thread is not the current thread a value of -1 is returned.

Thread debug status. The debug status of the thread.

0	The thread is disabled.
1	The thread is enabled.

Thread ID. This is an 8-byte thread handle assigned by the system.

Thread run state. The debug run status of the thread.

0	The thread is running.
1	The thread is currently stopped at a breakpoint, step, watch or unmonitored exception. When this happens all other threads are halted.
2	This is a thread that was halted by debug because of a debug stop that occurred in one of the debugged job's threads. The reason for stopping or halting all threads is to provide a static debugging environment.

Top of stack flag. Whether the stopped view ID is at the top of the call stack or not. Possible values are:

blank	This is not the current thread. This field is only applicable for the current thread.
0	The view ID is not at the top of the call stack.
1	The view ID is at the top of the call stack.

Error Messages

Message ID	Error Message Text
CPF3C19 E	Error occurred with receiver variable specified.
CPF3C1E E	Required parameter &1 omitted.
CPF3C21 E	Format name &1 is not valid.
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9549 E	Error addressing API parameter.
CPF958A E	Thread &1 not found.
CPF958B E	Threads debugging not supported.
CPF958C E	Number of threads not valid.
CPF958E E	Thread array special value not valid.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

API Introduced: V4R2

Top | Debugger APIs | APIs by category

Retrieve Module Views (QteRetrieveModuleViews) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	Format name	Input	Char(8)
4	Qualified program name	Input	Char(20) or Char(*)
5	Program type	Input	Char(10)
6	Module name	Input	Char(10) or Binary(4)
7	Returned library name	Output	Char(10) or Char(*)
8	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Retrieve Module Views (QteRetrieveModuleViews) API is used to return a list of modules and views associated with a specified program to the caller of the API. The list includes all of the following:

- All modules bound to the program that can be debugged
- Every view (by number and type) that was created by the compiler when the module object was created
- Views created by the OPM RPG, OPM COBOL, and OPM CL compilers using the *SRCDBG and *LSTDBG options
- Views created by the JAVA language support in OS/400

If you specify a module name, a list of views for that module is returned. If you specify *ALL for the module name, the list includes all modules for a given program.

The module name parameter must be specified as either *ALL or blanks for OPM programs. The statement view and a source view (or the statement view and a listing view) are always returned. The module name field is returned as blanks.

This API also supports JAVA class file debug views. In this case the program type parameter must be *CLASS and the qualified program name parameter must be a null-terminated JAVA class file name. The class path name of the file that contains the JAVA class file is returned in the returned library name parameter. For JAVA, the module name parameter must be specified as a binary field that contains the number of bytes provided in the returned library name field for JAVA class path name information.

Information returned by the Retrieve Module Views API is used by the calling program as input parameters to the Register Debug View API. Every module returned has at least one view associated with it. This is the statement view. It can be assumed that any additional views returned have text associated with them, and source debug can be done on these modules.

Authorities

The authorities required are dependent on the program type parameter. If the program type is *PGM or *SRVPGM, the authorities are as follows:

Program Authority

Either *SERVICE and *USE, or *CHANGE

Library Authority

*USE

If the program type is *CLASS, the authorities are as follows:

Class File Authority

*R

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

A variable that is to receive the information requested. You can specify the size of this area to be smaller than that needed to hold the information. In this case, only part of the information is returned. However, the number of bytes that the API needs to return all of the information is still returned.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. The minimum length is 8 bytes.

It is suggested that a length of 8 be passed to the API, which fills in the first two fields of the receiver variable. One of the fields, bytes available, indicates how much space must be provided. This space can then be obtained, and a second call to the API can be made.

Format name

INPUT; CHAR(8)

The content and format of the module view information that is returned. The only valid value for this parameter is:

VEWL0100	Module view information. For more information, see VEWL0100 Format.

Qualified program name

INPUT; CHAR(20) or CHAR(*)

The format of this parameter is dependent on the program type parameter. If the program type is *PGM or *SRVPGM, the format of this parameter is as follows:

- The name of a program about which module and view information is listed.
- The first 10 characters contain the program name.
- The second 10 characters contain the name of the library where the program can be located.
- Both entries must be left-justified.

The following special values may be used for the library name:

**CURLIB*	The job's current library.
**LIBL*	The library list.

If the program type is *CLASS, the format of this parameter is as follows:

The null-terminated class file name of the JAVA class.

Program type

INPUT; CHAR(10)

The type of program for which a view is to be registered. This is the object type of the program object. The allowable values are:

**PGM*	ILE or OPM program
**SRVPGM*	ILE service program
**CLASS*	JAVA class file name

Module name

INPUT; CHAR(10) or BINARY(4)

The format of this parameter is dependent on the program type parameter.If the program type is *PGM or *SRVPGM, the format of this parameter is as follows:

- A module name or *ALL (*ALL refers to all modules in the program).
- The module name parameter must be specified as either *ALL or blanks for OPM programs.

If the program type is *CLASS, the format of this parameter is as follows:

- A 4-byte binary field. This field contains the number of bytes provided in the returned library name parameter for returning JAVA class path name information.
- The value specified in this parameter must be at least 8 bytes.

Returned library name

OUTPUT; CHAR(10) or CHAR(*)

The format of this parameter is dependent on the program type parameter.If the program type is *PGM or *SRVPGM, the format of this parameter is OUTPUT CHAR(10) as follows:

- The library where the program was found. This is useful when *LIBL or *CURLIB is specified for the program library.

If the program type is *CLASS, the format of this parameter is OUTPUT CHAR(*) as follows:

- Class path name information for the requested class file. For more information, see Format of JAVA Returned Library Name Parameter.

Error code

I/O; CHAR(*)

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

VEWL0100 Format

The following table shows the format of the receiver variable for the VEWL0100 format.For more information on the fields, see Field Descriptions.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of elements
Note: The following fields repeat once for each element.			
		CHAR(10)	Module name
		CHAR(10)	View type
		CHAR(20)	Compiler ID
		CHAR(10)	Main indicator
		CHAR(13)	View timestamp
		CHAR(50)	View description
		CHAR(3)	Reserved
		BINARY(4)	View number
		BINARY(4)	Number of views

All views for a module are listed together in the receiver variable. The number of views field contains the total number of views for the module. The views are contiguous.

Field Descriptions

Bytes available. The number of bytes of data available to be returned to the user.

Bytes returned. The number of bytes of data returned to the user.

Compiler ID. The ID of the compiler that generated this view. For unique identification the first 4 bytes are used as follows:

x'00050000'	ILE C

x'00050001'	CSET C++ cooperative compiler
x'00060000'	ILE CL
x'00060001'	OPM CL
x'00070000'	OPM COBOL
x'00070001'	ILE COBOL
x'00170001'	OPM RPG
x'00170002'	ILE RPG
x'001D0000'	JAVA

Main indicator. Whether the module is a main module (entry point) for the program. The main indicator field can have one of the following values:

**MAIN*	Module is a main module
**NOMAIN*	Module is not a main module

There is at most one main module per program. Service programs contain no main entry point. *MAIN is always returned for OPM programs. For JAVA class files *MAIN is returned if the class file has a main procedure. Otherwise, *NOMAIN is returned for JAVA.

Module name. The name of the module for this list entry. For OPM programs and JAVA class files, the module name is returned as blanks.

Number of elements. The number of elements returned in the receiver variable. Each element has the same format, and it is repeated in the receiver variable. If the number of elements is zero and the receiver variable has room for at least one element, the program has no views in the module requested. If the module requested is *ALL, zero elements indicate the program cannot be debugged. For OPM programs, a CPF9584 error code is returned, instead of a zero number of elements value, if the program cannot be debugged. For class files, a CPF9599 error code is returned, instead of a zero number of elements value, if the program cannot be debugged.

Number of views. The number of views in this module listed in the receiver variable

Reserved. An ignored field.

View description. A character string that describes the view.

View number. A number that identifies a view within a module. Each view has a unique view number, which is used when you specify a specific view to register using the Register Debug View API.

View timestamp. The timestamp indicating when the view was created. It has the format of the American National Standard timestamp.

View type. The type of view. The view type can be one of the following values:

**TEXT*	This is a view where text comes from files or text supplied by the processor.
**LISTING*	This is a view where text comes entirely from text supplied by the processor.
**STATEMENT*	This is a view consisting of statement identifiers. All modules have a statement view.

Format of JAVA Returned Library Name Parameter

When the program type parameter is *CLASS, class path name information is returned in the returned library name parameter. The following table shows the format of the returned library name parameter when the JAVA class file view information is retrieved. For more information on the fields, see Field Descriptions.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Offset to class path name
C	C	BINARY(4)	Length of class path name
		CHAR(*)	Class path name

Field Descriptions

Bytes available. The number of bytes available to be returned in the returned library name parameter. If the bytes available value is larger than the bytes provided value passed in the module name parameter, the API should be called again with a value that is at least as large as the bytes available. If the space provided is not large enough, the string space is filled with as many characters of the class path name as will fit.

Bytes returned. The number of bytes returned in the returned library name parameter.

Class path name. The path name of the file that contains the class file that was retrieved.

Length of class path name. The length of the class path name returned.

Offset to class path name. The offset from the start of the returned library name parameter to the class path name.

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9549 E	Error addressing API parameter.
CPF954F E	Module &1 not found.
CPF955F E	Program &1 not a bound program.
CPF9584 E	OPM program &1 cannot be added to ILE debug environment.

CPF9585 E	Program &1 already active in OPM debug environment.
CPF9587 E	Module name value &1 not valid.
CPF9591 E	Value specified in module name parameter is not valid.
CPF9592 E	Class file not found.
CPF9593 E	Not authorized to class file.
CPF9594 E	JAVA class file not available.
CPF9599 E	Class file cannot be debugged.
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: V2R3

Retrieve Source Path Name (QteRetrieveSourcePathName) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	Format name	Input	Char(8)
4	Source file name	Input	Char(*)
5	Error code	I/O	Char(*)

Service Program: QTEDBGSI

Threadsafe: No

The Retrieve Source Path Name (QteRetrieveSourcePathName) API returns the full source path name for a Java source file.

This API expects the DEBUGSOURCEPATH environment variable to be set to one or more directory paths that contain Java source files. These directory paths are used to search for the Java source file specified by the source file name parameter. If the Java source file is found, the full source path name is returned in the receiver variable. If the Java source file is not found or if the DEBUGSOURCEPATH environment variable is not set, a CPF959E error is returned.

Authorities and Locks

Directory Authority

*X

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

A variable that is to receive the information requested. You can specify the size of this area to be smaller than that needed to hold the information. In this case, only part of the information is returned. The number of bytes that the API needs to return all of the information, however, is still returned.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. The minimum length is 8 bytes.

It is suggested that a length of 8 be passed to the API, which fills in the first two fields of the receiver variable. The bytes available field indicates how much space must be provided. This space can then be obtained and a second call can be made to the API.

Format name

INPUT; CHAR(8)

The content and format of the full source path information that is returned. The only valid value for this parameter is:

SRCP0100	Source path information. For more information, see SRCP0100 Format.

Source file name

INPUT; CHAR(*)

The null-terminated Java source file name (such as Hello.java).

Error code

I/O; CHAR(*)

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

SRCP0100 Format

The following table shows the format of the receiver variable for the SRCP0100 format. For more information on the fields, see Field Descriptions.

Offset		Type	Field
Dec	**Hex**		
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Offset to source path name
12	C	BINARY(4)	Length of source path name
		CHAR(*)	Source path name

Field Descriptions

Bytes available. The number of bytes of data available to be returned to the user.

Bytes returned. The number of bytes of data returned to the user.

Length of source path name. The length of the full source path name returned.

Offset to source path name. The offset from the start of the receiver variable to the source path name.

Source path name. The path name of where the Java source file resides. The Java source file name is returned as part of the source path name (for example, /home/javasource/Hello.java). The source path name is returned in the CCSID of the job.

Error Messages

Message ID	Error Message Text
CPF3C19 E	Error occurred with receiver variable specified.
CPF3C1E E	Required parameter &1 omitted.
CPF3C21 E	Format name &1 is not valid.
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF959E E	Source file not found.

API Introduced: V4R5

Set Debug Attribute (QteSetDebugAttribute) API

Required Parameter Group:

1	Debug attribute	Input	Char(10)
2	Attribute value	Input	Char(10)
3	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTEDBGS

Threadsafe: No

The Set Debug Attribute (QteSetDebugAttribute) API is used to set the attributes of the source debug session.

The attributes of the debug session cannot be set unless the job is currently in debug mode. The job is put in debug mode by a call to the Start Source Debug (QteStartSourceDebug) API.

The *UPDPROD value on the debug attribute parameter sets the update production files attribute of the debug session.

You can use files in production libraries while you are in debug mode. To prevent database files in production libraries from being changed unintentionally, you can specify a value of *NO. Then, only files in test libraries can be opened for updating or adding new records. If you want to open database files in production libraries for updating or adding new libraries, or if you want to delete members from production physical files, you can specify *YES. The initial setting when the Start Source Debug API is issued is *NO. However, this value can be changed at any time while in debug mode.

You can use this function with the library list. In the library list for your debug job, you can place a test library before a production library. In the test library, you should have copies of the production files that might be updated by the program being debugged. Then, when the program runs, it uses the files in the test library. Therefore, production files cannot be unintentionally updated.

The *OPMSRC value on the debug attribute parameter sets the OPM source debug attribute of the debug session. It is used to enable or disable the OPM source debug support. When this support is enabled, OPM RPG, OPM COBOL, and OPM CL programs can be debugged by using the ILE debug APIs if they were compiled with the *SRCDBG or *LSTDBG option on the following CL commands:

- Create RPG/400 Program (CRTRPGPGM)
- Create COBOL Program (CRTCBLPGM)
- Create Control language Program (CRTCLPGM)
- Create SQL RPG Program (CRTSQLRPG)
- Create SQL COBOL Program (CRTSQLCBL)

- Create Auto Report RPG Program (CRTRPTPGM)

The initial value of the *OPMSRC attribute is set by the Start Debug (STRDBG) command, and can also be changed by the Change Debug (CHGDBG) command. Changing the *OPMSRC value has no effect on programs that are already under debug. They remain in the debug environment (ILE or OPM) that they are currently added to.

Required Parameter Group

Debug attribute

INPUT; CHAR(10)

The name of the debug session that is to be set. The value of the debug attribute must be:

**UPDPROD*	Set the value of the update production files attribute.
**OPMSRC*	Set the value of the OPM source debug attribute.

Attribute value

INPUT; CHAR(10)

The value of the attribute specified in the debug attribute parameter.

When the debug attribute parameter specifies *UPDPROD, the attribute value parameter can have one of the following values:

**YES*	Allow the updating of production files while in debug mode.
**NO*	Do not allow the updating of production files while in debug mode.

When the debug attribute parameter specifies *OPMSRC, the attribute value parameter can have one of the following values:

**YES*	Allow OPM programs that have source debug data to be debugged by using the ILE debug APIs.
**NO*	Do not allow OPM programs to be debugged by using the ILE debug APIs.

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.

CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9549 E	Error addressing API parameter.
CPF9550 E	Value for debug attribute not valid.
CPF9559 E	Debug attribute parameter not valid.

API Introduced: V3R1

Top | Debugger APIs | APIs by category

Start Source Debug (QteStartSourceDebug) API

Required Parameter Group:

1	Qualified program name	Input	Char(20)
2	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTEDBGS

Threadsafe: No

The Start Source Debug (QteStartSourceDebug) API lets you use the source debugging APIs in your session. This allows the debugging of any ILE programs or service programs that contain debug information. OPM CL, OPM RPG, and OPM COBOL programs that are created with OPTION(*SRCDBG) or OPTION(*LSTDBG) may also be debugged.

Your job must be put in debug mode before this API is issued. Debug mode is a special environment in which the debug functions can be used in addition to routine system functions. Debug functions cannot be used outside debug mode. To start debug mode, you must issue the Start Debug (STRDBG) command.

The Start Source Debug API must be used before an ILE or OPM program can be debugged. This API requires that you specify a user exit program to be called by the source debugger support to handle breakpoints, steps, and unmonitored exceptions.

Your job remains in debug mode until an End Source Debug (QteEndSourceDebug) API is issued or until your current routing step ends.

If the job is servicing another job, the job will actually debug the job being serviced.

Authorities

Program Authority

*USE

Library Authority

*USE

Required Parameter Group

Qualified program name

INPUT; CHAR(20)

The name of the exit program that is called whenever a breakpoint, a program step, or an unmonitored exception occurs. See Program-Stop Handler Exit Program for a discussion of the parameters passed to this program to assist in processing breakpoint, step, and exception information.

The first 10 characters contain the program name. The second 10 characters contain the name of the library where the program is located. Both entries must be left-justified.

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.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9540 E	Already in debug mode.
CPF9541 E	Not in debug mode.
CPF9803 E	Cannot allocate object &2 in library &3.
CPF9809 E	Library &1 cannot be accessed.
CPF9810 E	Library &1 not found.
CPF9811 E	Program &1 in library &2 not found.
CPF9820 E	Not authorized to use library &1.
CPF9821 E	Not authorized to program &1 in library &2.
CPF9549 E	Error addressing API parameter.

API introduced: V2R3

Top | Debugger APIs | APIs by category

Stop Debugged Job (QteStopDebuggedJob) API

Required Parameter Group:			
1	Error code	I/O	Char(*)
Default Public Authority: *USE			
Service Program: QTETHRD			
Threadsafe: No			

The Stop Debugged Job (QteStopDebuggedJob) API causes debug to halt all threads in a job being debugged. The job stopped is being serviced and debugged by the job calling the QteStopDebuggedJob API. This API is allowed for servicing of both threaded and nonthreaded applications.

Required Parameter Group

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.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF958F E	Debug is not servicing a job.
CPF9590 E	Debugged job not stopped.

API Introduced: V4R2

Debug Session Handler Exit Program

Required Parameter Group:

1	Reason	Input	Char(10)
2	Program list	Input	Char(*)
3	Number of programs	Input	Binary(4)

Threadsafe: No

The Debug Session Handler exit program is a user-written program that manages the Integrated Language Environment (ILE) debugger. It determines when the debugger starts, stops, and shows its displays.

The name of the program is specified in the SRCDBGPGM parameter of the Start Debug (STRDBG) command. This program is called by the STRDBG command to initialize the user-written debugger, and is called by the End Debug (ENDDBG) command to end it. It is also called by the STRDBG and the Display Module Source (DSPMODSRC) commands to show the Display Module Source display.

If a JAVA class file name was specified in the JAVA parameter of the STRDBG command, the Debug Session Handler exit program will be called during debug initialization with a reason of *STARTJAVA. This call will be in addition to a separate call with a *START reason if ILE or OPM programs were also specified in the STRDBG PGM parameter.

Required Parameter Group

Reason

INPUT; CHAR(10)

The reason the program was called. Valid reasons include:

**START*	The program-stop handler should be initialized by the Start Source Debug API if this has not been done. The program list parameter format consists of 30-character entries. See the program list parameter description below.
**STARTJAVA*	The program-stop handler should be initialized by the Start Source Debug API if this has not been done. The program list parameter format consists of JAVA class file names. See the program list parameter description below.
**STOP*	The program-stop handler should be removed by the End Source Debug API.
**DISPLAY*	The debugger should display itself.
**RLSJOB*	The batch job being debugged has been released from the job queue. This is only supported for a debug session handler running in a batch job.

Program list

INPUT; CHAR(*)

The format of this parameter depends on the value of the reason parameter. If the reason parameter is *START, the program list format is as follows:

The list that is to receive a list of ILE or OPM programs to add to the debugger. This list contains the number of program entries, each entry being 30 characters in length. The first 10 characters contain the name of the program. The second 10 characters contain the name of the library where the program is located. The third 10 characters contain the type of object being named and can be *PGM (a callable program) or *SRVPGM (a service program). Each name is left-justified within the field.

If the reason parameter is *STARTJAVA, the format is as follows:

A list of JAVA class file name entries. For more information see Format of *STARTJAVA Program List Parameter. The number of list entries is contained in the number of programs parameter.

If the reason parameter is *DISPLAY, the format is as follows:

The eight character thread identifier of the current thread. This is valid only if threads debugging is allowed and the Number of programs parameter contains a value of 1.

If the reason parameter is *Stop or *RLSJOB, this parameter is not valid.

Number of programs

INPUT; BINARY(4)

The format of this parameter depends on the value of the reason parameter. If the reason parameter is *START or *STARTJAVA, the format is as follows:

The number of programs stored in the program list parameter.

If the reason parameter is *DISPLAY, the format is as follows:

The status of the threaded job. This is valid only if threads debugging is allowed.

0	The job is running and has not been stopped by debug (for example, breakpoint, step, watch, or unmonitored exception).
1	The job is stopped by debug.

If the reason parameter is *Stop or *RLSJOB, this parameter is not valid.

Format of *STARTJAVA Program List Parameter

When the reason parameter is *STARTJAVA the program list parameter contains JAVA class file names. The following table shows the format of the program list parameter for the *STARTJAVA reason. For more information on the fields, see Field Descriptions.

Offset		Type	Field
Dec	**Hex**		
Note: The following fields are repeated for each input class file name. The number of programs parameter contains the number of class file names.			
0	0	BINARY(4)	Offset to class file name
4	4	BINARY(4)	Length of class file name
Note: Following all of the above fields is a string space containing the input class file names.			
		CHAR(*)	Class file names

Field Descriptions

Class file names. The class file names that are specified on the STRDBG command.

Length of class file name. The length of the class file name.

Offset to class file name. The offset from the start of the program list parameter to the class file name.

Exit Program Introduced: V2R3

Program-Stop Handler Exit Program

Required Parameter Group:

1	Qualified program name	Input	Char(20) or Char(*)
2	Program type	Input	Char(10)
3	Module name	Input	Char(10)
4	Stop reason	Input	Char(10)
5	Receiver variable	Input	Char(*)
6	Number of entries	Input	Binary(4)
7	Message data	Input	Char(*)

QSYSINC Member Name: ETEPGMST

Threadsafe: No

The Program-Stop Handler exit program is a user-written program that handles program-stop conditions.

This program must be identified to the Source Debugger support with the Start Source Debug (QteStartSourceDebug) API.

Breakpoint- and step-program stop conditions are reported using stop reasons 2, 3, and 4. The location at which the program-stop condition occurred is specified in the receiver variable parameter and is in terms of the statement view. The user-supplied program may use the Map View Position (QteMapViewPosition) API to determine the location to which this program maps any other registered view.

Watch-program stop conditions are reported using stop reasons 5 and 6. For watch-program stop conditions, the program stopped might not have debug data. In this case, the machine interface (MI) number is reported for OPM programs and the statement number is reported for ILE programs and Java class files. If the program can be debugged, the line number in the statement view is reported for OPM programs, ILE programs, and Java class files. Other information is also included in the receiver variable to identify the program that caused the watch condition to be satisfied.

Unmonitored-exception-program stop conditions are represented through stop reason 1. Unmonitored exceptions are reported through this exit program only when the program and module identified have been created with debug data. Without debugging data, the message that is the cause of the unmonitored exception is issued, and the Program-Stop Handler exit program is not called.

When a job being debugged by a servicing job is stopped by the QteStopDebuggedJob API, reason code 7 is reported. When this reason code is reported, none of the other input parameters are used and can be ignored.

Debugging of threaded jobs is enabled by the thread ID field that is contained in the parameters passed to the stop handler. Threads debugging is supported if a service job is used to debug a job that was spawned by native threads support. For nonthreaded applications, the thread ID passed will always be that of the initial job thread.

Required Parameter Group

Qualified program name

INPUT; CHAR(20) or CHAR(*)

The format of this parameter is dependent on the program type parameter. If the program type is *PGM or *SRVPGM, the format of this parameter is as follows:

The name of the program that is stopped as a result of a breakpoint, program step, or unmonitored exception. This parameter may also be the name of the program that is stopped because a watch condition has been satisfied.

The first 10 characters contain the name of the program. The second 10 characters contain the name of the library where the program is located. Each name is left-justified.

If the program type is *CLASS, the format of this parameter is as follows:

The null-terminated class file name of the JAVA class.

Program type

INPUT; CHAR(10)

The object type of the program that is stopped. The possible values are:

**PGM*	Bound program or OPM program
**SRVPGM*	Service program
**CLASS*	JAVA class file

Module name

INPUT; CHAR(10)

The name of the module (left-justified) that is stopped. The value of this field is blank for OPM programs and JAVA class files.

Stop reason

INPUT; CHAR(10)

The reason the program was called. Each character of this parameter has a specific meaning. The characters and their meanings are:

1 This reason is set when an unmonitored exception is received by the program being serviced by the source debugger support.

0	No unmonitored exception
1	Unmonitored exception

2 The program stopped because an unconditional or conditional breakpoint was satisfied.

0	No break condition
1	Break condition

3	The program stopped because a step condition was reached.

0	No step condition
1	Step condition

4	The program stopped because a conditional breakpoint was set and there was a failure in running the condition. The program is stopped at the break position specified.

0	No break condition failure
1	Break condition failure

5	The program stopped because a watch condition set with the watch debug statement was satisfied.

0	No watch condition
1	Watch condition

6	The program stopped because there was a failure in processing the watch condition.

0	No watch condition failure
1	Watch condition failure

7	The debugged job being serviced was stopped by the QteStopDebuggedJob API.

0	Debugged job not stopped
1	Debugged job stopped

8-10	Reserved. These characters are set to 0.

Receiver variable

INPUT; CHAR(*)

Stop Reasons 1, 2, 3, 4:

If only stop reason 1, 2, 3, or 4 is present, the following receiver variable format is passed:

A list of locations within the statement view where the program stop condition occurred. This list contains the number of entries where each number is defined as follows:

Stopped locations	Array of BINARY(4) The line number in the statement view where the program is stopped.
Thread ID	CHAR(8) The thread identification of the thread where the program is stopped. This value immediately follows the last stopped location.

Stop Reasons 5, 6:

Whenever stop reason 5 or 6 is present (other stop reasons can be present also), the following receiver variable format is passed:

Information about the watch stop condition, including the program stopped and the program that caused the watch condition to be satisfied. See Format of Watch-Program Stop Reason for Receiver Variable.

Stop Reason 7:

For stop reason 7, the receiver variable parameter is not used and can be ignored.

Number of entries

INPUT; BINARY(4)

The number of positions stored in the receiver variable parameter. In some cases, it is not known exactly where a program is stopped; therefore, multiple positions are given. Each entry specifies one position in the statement view. This number is not less than one nor greater than three. At least one stopped position will be identified; if stopped at more than one position, no more than the first three positions are made available.

This parameter is valid when stop reason 1, 2, 3, or 4 is the only reason present (stop reason 5 or 6 cannot be present). If stop reason 5 or 6 is present, the receiver variable contains the equivalent number of stopped locations parameter.

Message data

INPUT; CHAR(*)

Information about the message. The information in this parameter is valid only when the stop reason specified is an unmonitored exception. For a detailed description of the format, see Format of Message Data.

Format of Watch-Program Stop Reason for Receiver Variable

The following table shows the information supplied in the receiver variable parameter when a stop reason of 5 or 6 is present. For more information on the fields, see Field Descriptions.

Watch Receiver Variable Header

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Watch number
4	4	BINARY(4)	Offset to stopped program information
8	8	BINARY(4)	Offset to watch interrupt information

Watch Stopped Program Information

The following table shows the stopped program information that is supplied in the receiver variable parameter. This data structure is accessible by adding the offset to stopped program information field in the receiver variable header to the address of the receiver variable.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Offset to stopped procedure name
4	4	BINARY(4)	Length of stopped procedure name
8	8	BINARY(4)	Offset to stopped locations
12	C	BINARY(4)	Number of stopped locations
16	10	CHAR(1)	Stopped locations flag
17	11	CHAR(3)	Reserved
20	14	CHAR(8)	Thread ID
		CHAR(*)	Reserved
		Array of BINARY(4)	Stopped locations
		CHAR(*)	Stopped procedure name

Watch Interrupt Information

The following table shows the watch-interrupt information that is supplied in the receiver variable parameter. This data structure is accessible by adding the offset to watch interrupt information field in the receiver variable header to the address of the receiver variable.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(26)	Qualified interrupt job name
26	1A	CHAR(20)	Qualified interrupt program name
46	2E	CHAR(10)	Interrupt program type
56	38	CHAR(10)	Interrupt module name
66	42	CHAR(1)	Interrupt locations flag
67	43	CHAR(1)	Reserved
68	44	BINARY(4)	Offset to interrupt procedure name
72	48	BINARY(4)	Length of interrupt procedure name
76	4C	BINARY(4)	Offset to interrupt locations
80	50	BINARY(4)	Number of interrupt locations
84	54	CHAR(8)	Thread ID
92	5C	BINARY(4)	Offset to interrupt class file name
96	60	BINARY(4)	Length of interrupt class file name
		CHAR(*)	Reserved
		Array of BINARY(4)	Interrupt locations

		CHAR(*)	Interrupt procedure name
		CHAR(*)	Interrupt class file name

Field Descriptions

Interrupt class file name. The Java class file name of the Java class containing the locations that caused the watch condition to be satisfied. For OPM and ILE programs, the Java class file name is not returned.

Interrupt locations. A list of locations, of the type described by the interrupt locations flag, that caused the watch condition to be satisfied.

Interrupt locations flag. The type of the locations in the interrupt locations field. All locations are of the same type.

1	Line number in statement view
2	Statement number
3	MI number

Interrupt module name. The name of the module (left-justified) in the program that caused the watch condition to be satisfied. The value of this field is blank for OPM programs and JAVA class files.

Interrupt procedure name. The procedure name of the procedure that contains the program locations that caused the watch condition to be satisfied. For OPM programs the procedure name is not returned.

Interrupt program type. The object type of the program that caused the watch condition to be satisfied. The possible values follow:

**PGM*	Bound program or OPM program
**SRVPGM*	Service program
**CLASS*	JAVA class file

Length of interrupt class file name. The length of the interrupt class file name. This field is zero if there is no interrupt class file name available (for example, OPM and ILE programs).

Length of interrupt procedure name. The length of the interrupt procedure name. This field is zero if there is no interrupt procedure name available (for example, OPM programs).

Length of stopped procedure name. The length of the stopped procedure name. This field is zero if there is no stopped procedure name available (for example, OPM programs).

Number of interrupt locations. The number of locations in the program that caused the watch condition to be satisfied. At most, three locations are returned.

Number of stopped locations. The number of stopped program locations. At most, three stop locations are returned.

Offset to interrupt class file name. The offset from the start of the receiver variable to the name of the Java class file containing the location that caused the watch condition to be satisfied. The field is zero if there is no interrupt class file name available (for example, OPM and ILE programs).

Offset to interrupt locations. The offset from the start of the receiver variable to the list of locations in the program that caused the watch condition to be satisfied.

Offset to interrupt procedure name. The offset from the start of the receiver variable to the name of the procedure that contains the program location that caused the watch condition to be satisfied. This field is zero if there is no interrupt procedure name available (for example, OPM programs).

Offset to stopped locations. The offset from the start of the receiver variable to the stopped program location entries.

Offset to stopped procedure name. The offset from the start of the receiver variable to the name of the procedure that contains the stopped program location. This field is zero if there is no stopped procedure name available (for example, OPM programs).

Offset to stopped program information. The offset from the start of the receiver variable to the stop information for the program that is stopped as a result of the watch condition being satisfied.

Offset to watch interrupt information. The offset from the start of the receiver variable to the watch interrupt information. This data structure describes the program that caused the interruption.

Qualified interrupt job name. The name of the job that caused the watch condition to be satisfied. The first 10 characters contain the job name. The second 10 characters contain the user profile name. The next 6 characters contain the job number. Each name is left-justified.

Note: This field is the same as the name of the job being debugged. Watch program interruptions in other jobs are ignored.

Qualified interrupt program name. The name of the program that caused the watch condition to be satisfied. The first 10 characters contain the name of the program. The second 10 characters contain the name of the library where the program is located. Each name is left-justified. The value of this field is blank for Java class files.

Reserved. An ignored field.

Stopped locations. A list of locations, of the type described by the stop location flag, where the program stop condition occurred.

Stopped locations flag. The type of the locations in the stop locations field. All stop locations are of the same type.

1	Line number in the statement view
2	Statement number
3	MI number

Stopped procedure name. The name of the procedure that contains the stopped locations. For OPM programs the procedure name is not returned.

Thread ID. This is an 8-byte thread identification that is assigned by the system. It identifies the thread associated with the stopped or interrupt locations fields.

Watch number. The watch number identifier of the watch condition being satisfied. This is the same number that is returned by the Submit Debug Command API when the watch condition was set.

Format of Message Data

The following table shows the information supplied in the message data parameter. For more information on the fields, see Field Descriptions.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Length of message data
4	4	CHAR(7)	Message ID
11	B	CHAR(20)	Message file
31	1F	CHAR(1)	Reserved
32	20	CHAR(512)	Message data

Field Descriptions

Length of message data. The length of the data available in the message data parameter, in bytes. This field contains the length of the available message data for the predefined message indicated in the message ID field.

Message data. The values for substitution variables in the predefined message specified in the message ID field and located in the message file field.

Message file. The name of the message file that contains the message identified in the message ID field.

The first 10 characters contain the message file name. The second 10 characters contain the name of the library where the file can be located. Both entries are left-justified.

Message ID. The identifying code of the message.

Reserved. An ignored field.

Exit program introduced: V2R3

»Service Entry Point Stop Handler Exit Program

Required Parameter Group:

1	Qualified program name	Input	Char(*)
2	Program type	Input	Char(10)
3	Module name	Input	Char(10)
4	Stop information	Input	Char(*)
5	Fully qualified job name	Input	Char(30)

QSYSINC Member Name: ETEPSEPH

Threadsafe: No

The **Service Entry Point Stop Handler exit program** is a user-written program that handles the service entry point program-stop condition.

This program must be identified to the Source Debugger support with the Register Service Entry Point Handler (QteRegSrvEntPntStpHdlr) API.

The location at which the service entry point was encountered is specified in the stop information parameter and is in terms of the statement view. The user-supplied program may use the Map View Position (QteMapViewPosition) API to determine the location to which this program maps any other registered view.

Debugging of threaded jobs is enabled by the thread ID field that is contained in the parameters passed to the stop handler. Threads debugging is supported if a service job is used to debug a job that was spawned by native threads support. For nonthreaded applications, the thread ID passed will always be that of the initial job thread.

Required Parameter Group

Qualified program name

INPUT; CHAR(*)

The format of this parameter is dependent on the program type parameter. If the program type is *PGM or *SRVPGM, the format of this parameter is as follows:

- The name of the program that is stopped as a result of a service entry point.
- The first 10 characters contain the name of the program. The second 10 characters contain the name of the library where the program is located. Each name is left-justified.

If the program type is *CLASS, the format of this parameter is as follows:

- The null-terminated class file name of the JAVA class.

Program type

INPUT; CHAR(10)

The object type of the program that is stopped.

The possible values are:

**PGM*	Bound program or OPM program
**SRVPGM*	Service program
**CLASS*	JAVA class file

Module name

INPUT; CHAR(10)

The name of the module (left-justified) that is stopped. The value of this field is blank for OPM programs and JAVA class files.

Stop information

INPUT; CHAR(*)

A list of locations within the statement view where the program stop condition occurred.

Thread ID	CHAR(8) The thread identification of the thread where the program is stopped.
Offset to stopped locations	BINARY(4) The offset from the start of the stop information to the first stop location.
Number of stopped locations	BINARY(4) The number of positions stored in the stop information parameter. In some cases, it is not known exactly where a program is stopped; therefore, multiple positions are given. Each entry specifies one position in the statement view. This number is not less than one nor greater than three. At least one stopped position will be identified; if stopped at more than one position, no more than the first three positions are made available.
Stopped locations	Array of BINARY(4) The line number in the statement view where the program is stopped.

Fully qualified job name

INPUT; CHAR(30)

The name of the job in which the program stop condition occurred. The fully qualified job name consists of three parts. The first 10 characters contain the job name. The next 10 characters contain the user name. The last 10 characters contain the 6-character job number followed by 4 blanks.

Exit program introduced: V5R2

Create View APIs

The create view APIs create view information, which is then available to source-level debugger applications through the source debugger APIs.

The create view APIs are:

- Add View Description (QteAddViewDescription) describes a view to be created.
- Add View File (QteAddViewFile) describes the files that can be used to construct the text for a view.
- Add View Map (QteAddViewMap) describes how to map positions in one view to positions in another view.
- Add View Text (QteAddViewText) describes the pieces of text making up the view text.
- End View Creation (QteEndViewCreation) completes view creation processing.
- Start View Creation (QteStartViewCreation) initializes the view creation environment.

Add View Description (QteAddViewDescription) API

Required Parameter Group:

1	Previous view number	Input	Binary(4)
2	View type	Input	Char(10)
3	Input/output	Input	Char(10)
4	Create map	Input	Char(10)
5	View description	Input	Char(50)
6	View number	Output	Binary(4)
7	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTECRTVS

Threadsafe: No

The program uses the Add View Description (QteAddViewDescription) API to add a new view in the existing view information. The added view can then be used on subsequent APIs when providing text and map details associated with this view.

It is the responsibility of each processor to create its input view, which is the root source file read by the processor. Each processor must also create its output view, which is the source produced by the processor. Other intermediate views may be produced, but, as a minimum, there must be a map between a processor's input and output view.

If a processor discards views produced by previous preprocessors, then it is not necessary for the input source view to be created. For example, the C compiler can create only a listing file view, as long as it discards all previous views.

It is possible to create several views at one time. It is the responsibility of the processes creating multiple views to manage them.

When a view is created, a handle to that view is returned in the form of a view number. This number is needed when adding text or maps that refer to the view. Once a view has been created, it cannot be created again. However, text and maps can be added to the view if it already exists. Thus, one processor can create the view, and another processor can add a map to the view, if that processor knows the view number.

There is only one statement view per module. If the statement view is created more than once, an error results. However, the statement view number is returned. This allows one processor to create the statement view and another processor to determine which number the view is.

Note: The following restrictions apply to the adding of views.

1. If a *TEXT view is added and that view refers to text in a previous view, the previous view must also be a *TEXT view.
2. The *INPUT and *OUTPUT views of a processor must be *TEXT views. A processor does not have to create these views.

Required Parameter Group

Previous view number

INPUT; BINARY(4)

The view number of a previous view whose text is used in creating the text for this view. When describing text for this view, it can be indicated that part of the text is a direct copy of text in the previous view. This allows the API to automatically generate a map between this view and the previous view.

As an example, if a preprocessor takes as input some source, changes it by expanding macros or SQL statements, and outputs the changed source, then the output view would have the input view as its previous view. When creating text from the output view, some of the text could come from the input view.

The previous view of a *TEXT view must also be a *TEXT view.

If there is no previous view, specify zero for the view number.

View type

INPUT; CHAR(10)

The type of view being created. Not all view types need be present in the view information. View type can be one of the following values:

**TEXT*	The view may contain supplied text as a result of macro expansions. Text may also come from a previous view or from files.
**LISTING*	Text for this view comes entirely from supplied text. Thus, the entire text for this view is encapsulated with the view debug data and is not dependent on the existence of source files.
**STATEMENT*	This view has no source text. Instead, the text of the view consists of HLL statement number, statement type, and the procedure dictionary ID. This view is necessary because breakpoint positions are given in terms of positions in this view.

Input/output

INPUT; CHAR(10)

Indicates whether the view is an input view, an output view, or an intermediate view. An input view is the view created from the output of the previous processor, or the view created from the root source file. It is not necessary for each processor to have an input view.

An output view is the view created by the processor to be input to the next processor. If a processor creates views that will not be used by any subsequent processors, then no output view is specified.

The allowable values for this parameter are:

**INPUT*	The view is an input view. This means that it must come from a root source file created by the user or by a previous processor, generally the input file specified on Start View Creation.

**OUTPUT*	The view is an output view. This means that it forms the text of a view that may be read by a subsequent processor, and is generally stored in the output file specified on Start View Creation.
blank	The view is neither an input nor an output view, but is an intermediate view produced by the processor.

Create map

INPUT; CHAR(10)

Specifies whether the using program will be supplying mapping information for this view, or whether the source debugger support should infer (create) the mapping at the time the text is described.

The purpose of the automatic mapping is to allow the ease of creating an include view. An include view has a previous view (usually the input view) which consists of only one file. The include view gets its text from this file and from include files.

This parameter applies only when the view type specified in the previous parameter is *TEXT, and when this view has a previous view. A map can then be inferred from the previous view to this view. To do so, the following criteria must be met:

- This view must contain text from the previous view whenever possible.
- The first file specified on the QteAddViewFile API call for this view must be the file which is equivalent to the previous view.
- When constructing the include view, the line with the include statement must never be included in the text of the view. Instead, it is replaced with the file that is specified.

Create map can be one of the following values:

**YES*	The source debugger support should infer the map between this *TEXT view and its previous view based on the text added with the QteAddViewText API. This is the only map to this view that is inferred.
**NO*	The program creating this *TEXT view uses the QteAddViewMap API to provide mapping information for this view.

View description

INPUT; CHAR(50)

A character string that describes the view being created. The source debugger has the option of displaying this text with the view for identification purposes. The description should be left-justified.

View number

OUTPUT; BINARY(4)

A number used to identify the view. Other APIs must be passed this number when they require a view.

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.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9547 E	Previous view not correct.
CPF9549 E	Error addressing API parameter.
CPF954B E	Statement view already exists.
CPF954D E	View type not valid.
CPF9555 E	Create Map parameter not valid.
CPF9556 E	API not valid at this time.
CPF955A E	Input Output parameter not valid.
CPF955D E	View data overflow. All debug data lost.

API Introduced: V3R1

Top | Debugger APIs | APIs by category

Add View File (QteAddViewFile) API

Required Parameter Group:

1	File descriptor buffer	Input	Char(*)
2	Number of entries	Input	Binary(4)
3	Format name	Input	Char(8)
4	View number	Input	Binary(4)
5	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTECRTVS

Threadsafe: No

The Add View File (QteAddViewFile) API provides a list of files that can be used when describing text for a previously added view. If a file is read more than once (such as a multiple included file), it should be added multiple times. When this file needs to be identified to other APIs, its file index is given, which is an index into the list of files supplied. The first file supplied has an index of zero.

The first file added to a view must be the root file for that view. For example, if a processor produces a source view, where a root file includes other files, the root file must be specified as the first file for the view. This is true even if the file is not the first file to produce view text (which would happen if an include statement is on the first line of the file).

All files for a view must be added at one time, with one call to this API.

Required Parameter Group

File descriptor buffer

INPUT; CHAR(*)

The input variable containing the list of files that make up the specified view text.

The source debugger support does not validate the existence of this file. This validation is done when text from the file is retrieved.

Number of entries

INPUT; BINARY(4)

The number of files that are provided in the file descriptor buffer parameter.

Many files may be described in a single file descriptor buffer. However, each entry must represent a single file, and this parameter must be a count of the number of files provided.

For format FILA0200, the number specifies the number of format entries, each containing seven fields, that are present before the external file names buffer.

Format name

INPUT; CHAR(8)

The content and format of the information supplied by the calling program in the file descriptor buffer. The valid values for format name are:

FILA0100	OS/400 file
FILA0200	External file (workstation file not on an iSeries server) or OS/400 integrated file system file

For more information, see FILA0100 Format or FILA0200 Format.

View number

INPUT; BINARY(4)

The number assigned by the debug support as an output parameter on the Add View Description API, which must be called prior to this API. If a file is used for more than one view, it must be supplied once for each view in which it is used.

Error code

I/O; CHAR(*)

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

FILA0100 Format

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(10)	OS/400 file name
10	A	CHAR(10)	OS/400 file library
20	14	CHAR(10)	OS/400 member name

FILA0200 Format

Offset			
Dec	**Hex**	**Type**	**Field**
Note: The first seven fields repeat the number of times specified in the number of entries parameter.			
		BINARY(4)	Offset of file name
		BINARY(4)	Length of file name
		BINARY(4)	File flag
		BINARY(4)	CCSID of file name
		CHAR(2)	Country or region ID of file name
		CHAR(3)	Language ID of file name

		CHAR(3)	Reserved
Note: The following field occurs after the preceding header fields.			
		CHAR(*)	File names buffer

Field Descriptions

OS/400 file library. The name of the library that contains the file being listed. It is an OS/400 object name, left-justified, and padded with blanks.

OS/400 file name. The name of the OS/400 file being listed. It is an OS/400 object name, left-justified, and padded with blanks.

OS/400 member name. The name of the member in the file being listed. It is an OS/400 object name, left-justified, and padded with blanks.

CCSID of file name. The CCSID the file name is in. A value of zero indicates to use the CCSID value of the job. A value of 65 535 causes an error message to be sent and the request to be ended.

Country or region ID of file name. The country or region ID of the file name. A value of blanks indicates that the country or region ID of job should be used.

File flag. Whether the file is an OS/400 integrated file system file or an external file (a workstation file not on an iSeries server).

0	External file
1	OS/400 integrated file system file

File names buffer. The names of external files or OS/400 integrated file system files being listed. The file names are laid out one after another in this buffer. There is a pair of offset and length fields for each file name in this buffer.

Language ID of file name. The language ID of the file name. A value of blanks indicates to use the language ID value of the job.

Length of file name. This is the length in bytes of the external file name in the external file names buffer.

Offset of file name. This offset from the start of the file descriptor buffer specifies the start of an external file name.

Reserved. Reserved for future use.

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.

CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF9556 E	API not valid at this time.
CPF9558 E	View already contains file descriptors.
CPF955B E	Number of entries not valid.
CPF955D E	View data overflow. All debug data lost.
CPF956B E	File name length not valid.
CPF956C E	File name offset not valid.
CPF9575 E	File flag not valid.
CPF9581 E	CCSID of file name parameter not valid.

API Introduced: V3R1

Top | Debugger APIs | APIs by category

Add View Map (QteAddViewMap) API

Required Parameter Group:

1	Map descriptor buffer	Input	Char(*)
2	Number of entries	Input	Binary(4)
3	Format name	Input	Char(8)
4	From view number	Input	Binary(4)
5	To view number	Input	Binary(4)
6	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTECRTVS

Threadsafe: No

The Add View Map (QteAddViewMap) API is used to map positions in one view to positions in another view. Both the view being mapped from and the view being mapped to must be previously added with the Add View Description API.

When mapping one view to another view, positions on both views are specified. There are two ways of specifying a position in the view:

- A line and column number in the view can be specified. This is the line number of the text of the view. The text is the concatenation of all text described by text descriptors.
- Alternatively, a file index, and a line and column in the file, can be specified. This allows view positions to be specified in terms of file positions.

A few rules must be followed when using the file method to specify positions. These rules pertain to the view for which file positions are to be specified:

1. The view must be a text view (type *TEXT).
2. All positions in the view must be specified using the file method. File positions and view positions may not be mixed.
3. The view may have a *TEXT view as a previous view, but the text from the previous view must consist of exactly one text descriptor, and that text descriptor must indicate that text comes from a file.

All lines in the from view must map to a position in the to view. For this reason, the first element in the mapping must specify line 1 of the from view.

This API is also used when mapping locations in a view to HLL statement numbers (the *STATEMENT view). This is accomplished by referencing the locations in the from view parameters (from view number and from line) and specifying the statement view number in the to view parameters (to view number and to line).

This API is also used when mapping HLL statement numbers (the statement view) to block numbers. This is accomplished by putting positions in the *STATEMENT view in the from line parameter and the block number in the to line parameter.

All the map positions for the two views must be added at once, with one call to this API. For this reason, the maps must be built up in a buffer as the processor produces its output. At the end of view creation, all maps are then supplied to this API.

Required Parameter Group

Map descriptor buffer

INPUT; CHAR(*)

The input variable containing view-mapping information that is to be passed to the API. This variable may contain multiple sets of information as long as each format in the variable is the same and the number of entries is specified appropriately.

Note: It is required that all map descriptors for a view be supplied with one call to this API.

Number of entries

INPUT; BINARY(4)

The number of map descriptors that are provided in the map descriptor buffer parameter.

Many map entries may be described in a single map descriptor buffer. However, each entry must represent a single map descriptor, and this parameter must be a count of the number of entries provided.

Each entry must contain all fields indicated, but, depending on the type of map being described by the entry, certain fields will not be used by the API.

Format name

INPUT; CHAR(8)

The content and format of the information supplied by the calling program in the map descriptor buffer. The valid values for format name are:

MAPA0100	Line and column mapping

From view number

INPUT; BINARY(4)

The number of the view being mapped from. This number must have been obtained from a previously added view with the Add View Description API.

Note: The from view number cannot refer to a statement view unless a map block is being created. In other words, a statement view cannot be mapped to a text view. However, text views may be mapped to the statement view.

To view number

INPUT; BINARY(4)

The number of the view being mapped to. This number must have been obtained from a view previously added with the Add View Description API. The special value supported is:

-1	This may be specified when providing a map from a *STATEMENT view to block identifiers.

Error code

I/O; CHAR(*)

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

MAPA0100 Format

The following table shows the information supplied in the MAPA0100 format:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	From file index
4	4	BINARY(4)	From line
8	8	BINARY(4)	From column
12	C	BINARY(4)	To file index
16	10	BINARY(4)	To line
20	14	BINARY(4)	To column
24	18	BINARY(4)	Map type

Field Descriptions

From column. The column number where text is located. This is the character position on the line specified above. Column numbers in the range 1 through 255 can be specified.

Note: Column numbers are not supported at this time. For upward compatibility, you should specify a value of 1.

From file index. The number of the file, if the position is specified in terms of a file. This file must have been added and used in a text descriptor of the view. If the position is specified in terms of a view, the file index must be set to special value -1.

When file 0 is specified and the view has a previous view, then this file is assumed to mean a line in the previous view, because the first file specified in the file descriptor must be the root source file used to construct this view. This root source file is the same as the previous view.

From line. The line number of the view or file where the text is mapped from.

When the from view is a *STATEMENT view, this parameter indicates which statement to map from. Its position in the *STATEMENT view is supplied, the first statement having position 1.

Map type. Specifies how text from the from view is being mapped to the to view at the position indicated.

0	The type is not supplied, and is allowed only for statement or block mappings where the type is always known.
1	The text from the from view is being copied to the to view at the specified positions.

2 The text in the to view is an expansion of text in the from view at the specified positions. This is because of a macro expansion or an include statement.

To column. The column number specifying the starting character position within the line where the text is to go. Column numbers in the range 1 through 255 can be specified.

> **Note:** Column numbers are not supported at this time. For upward compatibility, you should specify a value of 1.

To file index. The number of the file, if the position is specified in terms of a file. This file must have been added and used in a text descriptor of the view. If the position is specified in terms of a view, the file index must be set to special value -1.

When file 0 is specified and the view has a previous view, then this file is assumed to mean a line in the previous view, because the first file specified in the file descriptor must be the root source file used to construct this view. This root source file is the same as the previous view.

To line. The line number of the view or file where the text is mapped to.

When the to view is a *STATEMENT view, this parameter indicates which statement to map to, the first view having position 1.

When the to view is a block view (special value -1), this parameter indicates which block number to map to.

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9543 E	From view not found.
CPF9544 E	To view not found.
CPF9549 E	Error addressing API parameter.
CPF9551 E	File not found.
CPF9552 E	Cannot map between views.
CPF9553 E	Map type not defined.
CPF9556 E	API not valid at this time.
CPF955B E	Number of entries not valid.
CPF955D E	View data overflow. All debug data lost.

API Introduced: V3R1

Add View Text (QteAddViewText) API

Required Parameter Group:

1	View number	Input	Binary(4)
2	Text descriptor buffer	Input	Char(*)
3	Number of entries	Input	Binary(4)
4	Format name	Input	Char(8)
5	Supplied text buffer	Input	Char(*)
6	Length of text buffer	Input	Binary(4)
7	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTECRTVS

Threadsafe: No

The Add View Text (QteAddViewText) API is used to describe a piece of text of a previously added view. As a processor reads its input source, it creates at least one view. This API is issued to add the directions for re-creating the text of these views. For the debugger to show the text that makes up a view, the location of the pieces of text that make up the view must be specified.

When the view is reconstructed by the debugger, the pieces of text will be retrieved and concatenated into a single piece of text, following the directions given when this API is called. Thus, when it is mentioned that text is copied, it is referring to a later time, when the view is reconstructed.

All the text for a view must be added at once, with one call to this API. For this reason, the text must be built up in a buffer as the processor produces its output. At the end of view creation, all text is then supplied to this API.

If any text comes from files, the file descriptors must have been previously added to the view with the Add View File (QteAddViewFile) API.

Required Parameter Group

View number

INPUT; BINARY(4)

The number of the view to which a piece of text is being added. This number must be the same as the number previously returned by the Add View Description API.

Text descriptor buffer

INPUT;CHAR(*)

The input variable containing the text descriptors. Text descriptors define the location of text used to build the view specified in the view name parameter.

Number of entries

INPUT;BINARY(4)

The number of text descriptors that are provided in the text descriptor buffer parameter.

Many pieces of text may be described in a single text descriptor buffer. However, each entry must represent a single piece of contiguous text, and this parameter must be a count of the number of entries provided.

Each entry must contain all fields indicated, but, depending on the type of text being described by the entry, certain fields will not be used by the API.

If any text is supplied by the calling program, it is identified by a text descriptor, but the text itself is contained in the supplied text buffer.

Format name

INPUT; CHAR(8)

The content and format of the information supplied by the calling program in the text descriptor buffer. The valid values for format name are:

TXTA0100	Used when the text being added to this view can come from any of the following: ❍ Blanks ❍ Stored in a file ❍ Copied from the previous view ❍ Supplied by the calling program within the supplied text buffer parameter This is the case for a *TEXT view.
TXTA0101	Used when the entire text for this view is supplied text. This is the case for a *LISTING view.
TXTA0102	Used when statement information for a *STATEMENT view is to be supplied.
TXTA0103	Used when the entire text for this view is supplied text. This is the case for a *LISTING view. Note that this format is identical to TXTA0101; however, when the TXTA0103 format is specified, the listing view that is created will be compressed. When it is reconstructed, it will be decompressed at that time.

Supplied text buffer

INPUT; CHAR(*)

The input variable that is to be passed to the API, containing the text that is supplied when a text descriptor in the text descriptor buffer parameter indicates that text is supplied. Text descriptors within the text descriptor buffer refer to text locations within this buffer by the offset from the beginning of the buffer. The piece of text is ended by a NULL (hex 00) character. To conserve storage, delete the trailing blanks in lines of supplied text, and end the text with a null character.

Note: A line of supplied text must not be more than 255 characters in length, not counting the NULL character.

Length of text buffer

INPUT; BINARY(4)

The length of the supplied text buffer.

Error code

I/O; CHAR(*)

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

TXTA0100 Format

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(10)	Text location
10	A	CHAR(2)	Reserved
12	C	BINARY(4)	File index
16	10	BINARY(4)	Starting offset
20	14	BINARY(4)	Number of lines
24	18	BINARY(4)	From line

TXTA0101 Format

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Starting offset

TXTA0102 Format

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Procedure dictionary ID
4	4	BINARY(4)	Statement number
8	8	CHAR(1)	Statement type

TXTA0103 Format

Offset			
Dec	**Hex**	**Type**	**Field**

0	0	BINARY(4)	Starting offset

Field Descriptions

File index. A file member added by the Add View File API. This field is required if the text location is set to *FILE; otherwise, it is ignored. The first file added for the specified view is file 0, the second is file 1, and so forth.

When the view has a previous view, file 0 should not be specified. File 0 is assumed to mean a line in the previous view because the first file specified in the file descriptor must be the root source file used to construct this view. This root source file is the same as the previous view. Instead, *PREVIOUS should be specified in the text location field. If file 0 is specified instead of the previous view and the previous view was created by another preprocessor that created a temporary file as its output, that file may not exist at run time. In that case, text for the view could not be retrieved. However, if *PREVIOUS is specified, the View Retrieval API can use the text descriptors of the output view created by the preprocessor to reconstruct the text.

> **Note:** The source debugger support does not validate the existence of this file. It merely uses the name in the view information to refer to the location of debug data. When the text of the view is reconstructed, text will be retrieved from the file named in this parameter (and the member name parameter), and the file name will be validated at that time.

From line. The line number where text is located. If the text location is a file, this is the line number in that file. If the text location is a previous view, this is the line position within that view. This can be thought of as the start line position. This field is required if the text location is set to *FILE or *PREVIOUS.

Number of lines. The number of lines of text being described. It is intended that views be created in order, where each piece of text comes directly after the previous text added. This field is required when text location is set to *FILE or *PREVIOUS.

Procedure dictionary ID. The dictionary number of the procedure where the statement is located.

Reserved. Reserved for future use.

Starting offset. The location within the supplied text buffer of the start of the supplied text. This is an offset from the beginning of the buffer to the start of the text. This field is required if the text location is set to *SUPPLIED in the TXTA0100 format.

Statement number. The HLL statement number of the statement.

Statement type. The type of the statement being added. Possible values are:

X'01'	INIT CODE
X'02'	PROC ENTRY
X'03'	PROC EXIT
X'04'	ALLOC
X'05'	STMT
X'06'	ENTRY
X'07'	EXIT
X'08'	MULTIEXIT

X'09'	PATH LABEL
X'10'	PATH CALL BGN
X'11'	PATH CALL RET
X'12'	PATH DO BGN
X'13'	PATH TRUEIF
X'14'	PATH FALSEIF
X'15'	PATH WHEN BGN
X'16'	PATH OTHERW
X'17'	GOTO
X'18'	POST COMPOUND

Text location. The location of the text being referred to. This field is required for all entries.

**FILE*	The text is stored in a file.
**PREVIOUS*	The text is a copy of the previous view text. The previous view is specified when the view is created.
**SUPPLIED*	The text is supplied by the API user within the supplied text buffer parameter. The text that is supplied by the using program must be in the suppled text buffer parameter and referred to by a text descriptor within the text descriptor buffer parameter.
**BLANK*	The text consists of blank lines. The number of blank lines inserted is specified by the number of lines field.

Error Messages

Message ID	**Error Message Text**
CPF3C21 E	Format name &1 is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9542 E	View not found.
CPF9545 E	No previous view.
CPF9549 E	Error addressing API parameter.
CPF954E E	Text location is not valid.
CPF9551 E	File not found.
CPF9552 E	Cannot map between views.
CPF9556 E	API not valid at this time.
CPF9557 E	View already contains text descriptors.
CPF955B E	Number of entries not valid.
CPF955C E	Supplied Text Length parameter not valid.
CPF955D E	View data overflow. All debug data lost.

CPF9569 E	Missing supplied text.
CPF956A E	No such text in previous view.

API Introduced: V4R2

Top | Debugger APIs | APIs by category

End View Creation (QteEndViewCreation) API

Required Parameter Group:			
1	Error code	I/O	Char(*)
Default Public Authority: *USE			
Service Program: QTECRTVS			
Threadsafe: No			

The End View Creation (QteEndViewCreation) API is used by a processor when all debug data views have been created. At that time, views are written to the output file member (if any) specified on the Start View Creation API. This End View Creation API should not be called if the view is not complete (for example, if a compiler that is creating the view fails the compilation).

Authorities

Library Authority
    *USE
File Authority
    *CHANGE

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
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9546 E	View information damaged.
CPF9549 E	Error addressing API parameter.
CPF9556 E	API not valid at this time.

CPF955D E	View data overflow. All debug data lost.
CPF9803 E	Cannot allocate object &2 in library &3.
CPF9809 E	Library &1 cannot be accessed.
CPF9810 E	Library &1 not found.
CPF9815 E	Member &5 file &2 in library &3 not found.
CPF9820 E	Not authorized to use library &1.
CPF9822 E	Not authorized to file &1 in library &2.

API Introduced: V3R1

Top | Debugger APIs | APIs by category

Start View Creation (QteStartViewCreation) API

Required Parameter Group:

1	Input file descriptor buffer	Input	Char(*)
2	Output file descriptor buffer	Input	Char(*)
3	Format name	Input	Char(8)
4	Discard previous views	Input	Char(10)
5	Processor ID	Input	Char(20)
6	View CCSID	Input	Binary(4)
7	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTECRTVS

Threadsafe: No

The calling program uses the Start View Creation (QteStartViewCreation) API to initialize the debug view creation environment. This API should be the first one of the view creation APIs to be called.

This calling program is usually a text preprocessor or a compiler. In this document, the term processor will be used to specify any program that reads input text and produces view data for the debugger.

The processor that calls the Start View Creation API must provide the names of the input source member read and the output source member created (if any).

The input member name is the name of the root source member read by the processor. If a previously run processor created this member, then view information is stored with the member. This view information is combined with that produced by the processor and stored in the output source member specified by the processor.

If an input member is specified and there is no view information in the associated space of the member, it is assumed that this member is the root source from which the program is created. It is also assumed that the processor that specifies this member is the first processor in the chain of processors that produces the program.

The processor creating the view supplies the CCSID in which all supplied text is stored. Thus, when view text is extracted, all supplied text is translated from this CCSID to the CCSID of the job. When view text (such as macro expansion text) is supplied, it must be supplied in the same CCSID. Text that comes from other files may be in any CCSID, as it will automatically be translated into the job's CCSID when the text is retrieved.

If no input member is specified, it is assumed that a previous processor created view information in a temporary space, instead of storing it in an output member. This is the case when a compiler runs and produces view information. Since the compiler does not produce an output member to be read by another processor, the view information is stored in a temporary location associated with the process, and no output file is specified.

Each processor creates view information that is combined with information produced by previous processors. The final, and complete, view information is stored by the binder in the module and program object associated space.

After the view information is complete, the End View Creation API should be called.

The input file must exist and be available at the time this API is issued. The output file must exist and be available at the time the QteEndViewCreation API is issued.

Authorities

Input file member specified

*USE

Output file member specified

*CHANGE

Required Parameter Group

Input file descriptor buffer

INPUT; CHAR(*)

The name of the file member read by the processor creating debug data. This member may be a member created by the customer, or it may be the output of a previously run preprocessor. The special value of *NONE is used when input from the processor does not come from a source member. In general, the only processor which would indicate *NONE is the back end of the compiler.

This file may contain view information if it is created by a previously run preprocessor.

The structure of this parameter is specified by the format name parameter.

Output file descriptor buffer

INPUT; CHAR(*)

The file member written by the processor creating debug data. A special value of *NONE for the output file indicates that the view information created will remain with the job and will be passed to the next compilation step without being associated with a specific file. Generally, only the compiler uses this special value, as it does not create a source member to be read by another processor.

The associated space of this file will contain view information created by this processor in addition to view information created by any previous preprocessor steps. It is the responsibility of the processor to create this file and make it available before the QteEndViewCreation API is called.

The structure of this parameter is specified by the format name parameter.

Format name

INPUT; CHAR(8)

The content and format of the information supplied by the calling program in the input file descriptor buffer and the output file descriptor buffer. The only valid value for format name is:

FILA0100	OS/400 file names

Discard previous views

INPUT; CHAR(10)

Whether the program creating debug view information wants the source debugger support to throw away any previously created view information. This allows a processor to force the view information created to be the only debug data available. In general, processors would specify *NO to allow any previous processor's view information to be included in the final program object.

**YES*	The source debugger support does not use any previously built view information but rather starts with the information provided by the processor creating debug data.
**NO*	The source debugger support uses any previously built and existing view information and adds to it the view information created during this compiler step.

Processor ID

INPUT; CHAR(20)

The processor that creates view information.

View CCSID

INPUT; BINARY(4)

The CCSID of any text supplied to the view creation APIs.

Error code

I/O; CHAR(*)

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

FILA0100 Format

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(10)	File name
10	A	CHAR(10)	File library
20	14	CHAR(10)	Member name

Field Descriptions

File library. The name of the library that contains the file being listed. It is an OS/400 object name, left-justified, and padded with blanks.

File name. The name of the file being listed. It is an OS/400 object name, left-justified, and padded with blanks.

Member name. The name of the member in the file being listed. It is an OS/400 object name, left-justified, and padded with blanks.

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9549 E	Error addressing API parameter.
CPF9554 E	Discard Previous Views parameter not valid.
CPF9803 E	Cannot allocate object &2 in library &3.
CPF9809 E	Library &1 cannot be accessed.
CPF9810 E	Library &1 not found.
CPF9815 E	Member &5 file &2 in library &3 not found.
CPF9820 E	Not authorized to use library &1.
CPF9822 E	Not authorized to file &1 in library &2.

API Introduced: V3R1

View Information APIs

View information APIs retrieve view information, including view text information and view mapping information, and allow the program to set parameters associated with a view.

The view information APIs are:

- Map View Position (QteMapViewPosition()) used to map positions in one view to positions in another view.
- Register Debug View (QteRegisterDebugView) registers a view of a module, which allows a program to be debugged in terms of that view.
- Remove Debug View (QteRemoveDebugView) removes a view of a module that was previously registered by the Register Debug View API. This is necessary when a program is to be removed from debug mode so it can be deleted and recompiled.
- Retrieve Statement View (QteRetrieveStatementView) returns the statement view and related information.
- Retrieve Stopped Position (QteRetrieveStoppedPosition) determines if a program is on the call stack and indicates the position in the view at which the program is stopped if it is on the stack.
- Retrieve View File (QteRetrieveViewFile) returns all the files and text information necessary to construct the text for a view.
- Retrieve View Line Information (QteRetrieveViewLineInformation) returns information about the specified number of lines in a registered view.
- Retrieve View Text (QteRetrieveViewText()) retrieves the text of a view.

Map View Position (QteMapViewPosition) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	From view ID	Input	Binary(4)
4	From line number	Input	Binary(4)
5	From column number	Input	Binary(4)
6	To view ID	Input	Binary(4)
7	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTEDBGS

Threadsafe: No

The Map View Position (QteMapViewPosition) API maps positions from one view to another view within the same program and module. A specified position in the view identified in the from view ID parameter is used for the mapping. The position is specified as a line number and a column number in the from view ID.

A position in one view can map to more than one position in another view. For example, an SQL statement in the SQL input source view may map to many positions in the C input source view.This is because a single SQL statement may distribute source throughout the output of the SQL processor.

One or more positions in the to view ID are returned as a line number and a column number.

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The variable that is to receive the information requested. You can specify the size of this area to be smaller than the format requested if you specify the length of receiver variable parameter correctly. As a result, the API returns only the data that the area can hold. For more information on the size and format of the receiver variable, see Format of Receiver Variable.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. The minimum length is 8 bytes.

It is suggested that a receiver variable length be given that is large enough to hold one map element. Because this is normally the number of elements returned, a single call to this API is usually sufficient.

From view ID

INPUT; BINARY(4)

The identifier of a previously registered view, which is obtained using theRegister Debug View API. This ID specifies the from view in the mapping function provided.

From line number

INPUT; BINARY(4)

The line number in the view specified by the from view ID parameter mapped to aline number in the view specified by the to view ID parameter.

If the information in the from view ID parameter is a statement view, this parameter represents the line number in the statement view.

Note: The statement view is the lowest level view. Breakpoints, steps, and unmonitored exceptions are reported as a line number within this view. Therefore, the statement view must exist and be registered to accomplish source level debugging.

From column number

INPUT; BINARY(4)

The position in the line specified by the from line number parameter. Column numbers are 1 through 255.

If the from view ID parameter is a statement view, this parameter is not used and should be set to column one.

To view ID

INPUT; BINARY(4)

The identifier of a previously registered view, which is obtained using the Register Debug View API. This specifies the to view in the mapping function provided.

Error code

I/O; CHAR(*)

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

Format of Receiver Variable

The following table shows the information supplied in the receiver variable parameter.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of map elements
Note: The following fields are repeated for each map element.			
		BINARY(4)	Line number
		BINARY(4)	Column number

Field Descriptions

Bytes available. The number of bytes of data available to be returned to the user.

Bytes returned. The number of bytes of data returned to the user.

Column number. The column number within the from line number parameter that maps to the current position in the to view ID parameter. Column numbers are 1 through 255.

If the view is a statement view, this number is not used and is set to column one.

Line number. The line number in the view specified by the to view ID parameter.

Number of map elements. Theline number and column number fields are repeated this number of times, once for each map available.

Error Messages

Message ID	Error Message Text
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9543 E	From view not found.
CPF9544 E	To view not found.
CPF9548 E	Map not available.
CPF9567 E	Column number not valid.
CPF9568 E	Line number not valid.
CPF9549 E	Error addressing API parameter.

API Introduced: V2R3

Register Debug View (QteRegisterDebugView) API

Required Parameter Group:

1	View ID	Output	Binary(4)
2	Number of lines	Output	Binary(4)
3	Returned library	Output	Char(10) or Char(*)
4	View timestamp	Output	Char(13)
5	Qualified program name	Input	Char(20) or Char(*)
6	Program type	Input	Char(10)
7	Module name	Input	Char(10) or Binary(4)
8	View number	Input	Binary(4)
9	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Register Debug View (QteRegisterDebugView) API registers a view of a module, which allows a program to be debugged in terms of that view. An identifier to the view is returned on successful completion of the API to be used in subsequent view information APIs. A program is considered to be active under ILE debug only after at least one of its debug views is registered.

Views retrieved by the Retrieve Module Views (QteRetrieveModuleViews) API can be registered. This includes both ILE and OPM program views. OPM program views must have been created by the OPM CL, OPM COBOL, or OPM RPG compiler using the *SRCDBG or *LSTDBG option.

This API will also register JAVA class file views. In this case the input program type parameter must be *CLASS and the input qualified program name parameter must be a null-terminated JAVA class file name. The class path name of the file that contains the class file is returned in the returned library parameter.

If a request is made to register an already registered view, no error occurs. Instead, the previous ID is returned.

Note: Before registering views for a program again, it is recommended that all views for that program first be removed.

Authorities

The authorities required are dependent on the program type parameter. If the program type is *PGM or *SRVPGM, the authorities are as follows:

Program Authority

Either *SERVICE and *USE, or *CHANGE

Library Authority

*USE

If the program type is *CLASS, the authorities are as follows:

Class File Authority

*R

Required Parameter Group

View ID

OUTPUT; BINARY(4)

The returned ID of the successfully (or previously) registered debug view. The source debugger support supplies and maintains the view IDs. If no error is reported by the API, this value is used by the program in view ID input parameters that occur on subsequent debugger APIs.

Number of lines

OUTPUT; BINARY(4)

The number of lines of text in the view.

Returned library

OUTPUT; CHAR(10) or CHAR(*)

The format of this parameter is dependent on the program type parameter. If the program type is *PGM or *SRVPGM, the format of this parameter is OUTPUT CHAR(10) as follows:

The library where the program was found. This is useful when *LIBL or *CURLIB is specified for the program library.

If the program type is *CLASS, the format of this parameter is OUTPUT CHAR(*) as follows:

Class path name information for the requested class file. For more information, see Format of JAVA Returned Library Parameter.

View timestamp

OUTPUT; CHAR(13)

The date and time the view was created. If this time is greater than the time obtained from the Retrieve Module Views API, the view may not be the same as the previous one. Users should run the Retrieve Module Views API before registering the view. The value is the American National Standard 13-character timestamp CYYMMDDHHMMSS format, where:

C	Century, where 0 indicates years 19*xx* and 1 indicates years 20*xx*.
YY	Year
MM	Month
HH	Hour
MM	Minute
SS	Second

Qualified program name

INPUT; CHAR(20) or CHAR(*)

The format of this parameter is dependent on the program type parameter. If the program type is *PGM or *SRVPGM, the format of this parameter is as follows:

- The name of a program for which a view is to be registered.
- The first 10 characters contain the program name.
- The second 10 characters contain the name of the library where the program is located.

  The following special values may be used for the library name:

**CURLIB*	The job's current library.
**LIBL*	The library list.

If the program type is *CLASS, the format of this parameter is as follows:

The null-terminated class file name of the JAVA class to register.

Program type

INPUT; CHAR(10)

The type of program for which a view is to be registered. This is the object type of the program object. The valid values are:

**PGM*	ILE or OPM program
**SRVPGM*	ILE service program
**CLASS*	JAVA class file

Module name

INPUT; CHAR(10) or BINARY(4)

The format of this parameter is dependent on the program type parameter. If the program type is *PGM or *SRVPGM, the format of this parameter is as follows:

- The name of a module for which a view is to be registered.
- The module name should be left-justified.
- The module name parameter must be specified as all blanks for OPM programs.

Information for this parameter is available by using the Retrieve Module Views API to retrieve available module names for a specified program.

If the program type is *CLASS, the format of this parameter is as follows:

- The module name parameter must contain a 4-byte binary field.
- This field contains the number of bytes provided in the returned library parameter for returned JAVA class path name information.
- The value specified in this parameter must be at least 8 bytes.

View number

INPUT; BINARY(4)

The number of a view to be registered for subsequent view information and debug command APIs. If -1 is specified, the statement view is registered. The value -1 is a shortcut to allow the registering of this view without going through the Retrieve Module Views API to obtain the number.

Information for this parameter is available by using the Retrieve Module Views API to retrieve available view numbers for modules associated with a specific program.

Error code

I/O; CHAR(*)

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

Format of JAVA Returned Library Parameter

When the program type parameter is *CLASS, class path name information is returned in the returned library parameter. The following table shows the format of the returned library parameter when JAVA class file view information is registered. For more information on the fields, see Field Descriptions.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Offset to class path name
C	C	BINARY(4)	Length of class path name
		CHAR(*)	Class path name

Field Descriptions

Bytes available. The number of bytes available to be returned in the returned library parameter. If the bytes available value is larger than the bytes provided value passed in the module name parameter, the API should be called again with a value that is at least as large as the bytes available. If the space provided is not large enough, the string space is filled with as many characters of the class path name as will fit.

Bytes returned. The number of bytes returned in the returned library parameter.

Class path name. The path name of the file that contains the class file that was retrieved.

Length of class path name. The length of the class path name returned.

Offset to class path name. The offset from the start of the returned library parameter to the class path name.

Error Messages

Message ID	Error Message Text
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
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.
CPF9584 E	OPM program &1 cannot be added to ILE debug environment.
CPF9585 E	Program &1 already active in OPM debug environment.
CPF9587 E	Module name value &1 not valid.
CPF9588 E	OPM source cannot be accessed.
CPF9591 E	Value specified in module name parameter is not valid.
CPF9592 E	Class file not found.
CPF9593 E	Not authorized to class file.
CPF9594 E	JAVA class file not available.
CPF9599 E	Class file cannot be debugged.
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: V2R3

Remove Debug View (QteRemoveDebugView) API

Required Parameter Group:

1	View ID	Input	Binary(4)
2	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Remove Debug View (QteRemoveDebugView) API removes a view of a module that was previously registered by the Register Debug View API. This API is needed when a program is to be removed from debug, so that it can be deleted and recompiled. Once a view is removed from being debugged, its view number may not be used again.

If the last registered view of a program is removed, all breakpoints are removed from that program, and the step statement is disabled if it was active.

Required Parameter Group

View ID

INPUT; BINARY(4)

The ID of a view to be removed from debug. This ID was obtained from a previous use of the Register Debug View API.

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.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.

API Introduced: V2R3

Retrieve Statement View (QteRetrieveStatementView) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	View ID	Input	Binary(4)
4	Start line number	Input	Binary(4)
5	Number of lines	Input	Binary(4)
6	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Retrieve Statement View (QteRetrieveStatementView) API is used to retrieve the statement view and related information. The statement view information that is retrieved can be useful for breakpoint processing. The caller must specify the following:

- The registered statement view ID
- The starting statement view line number to be retrieved
- The number of statement view lines to retrieve
- A buffer to contain the statement view and related information

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The receiver variable that receives the information requested. You can specify the size of the area to be smaller than the format requested as long as you specify the length parameter correctly. As a result, the API returns only the data that the area can hold. For more information, see Format of Receiver Variable.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable provided. The length of receiver variable parameter may be specified up to the size of the receiver variable specified in the user program. If the length of receiver variable parameter specified is larger than the allocated size of the receiver variable specified in the user program, the results are not predictable. The minimum length is 8 bytes.

View ID

INPUT; BINARY(4)

The identifier of the previously registered statement view obtained by using the Register Debug View (QteRegisterDebugView) API.

Start line number

INPUT; BINARY(4)

The number of the first statement view line that the API is to retrieve. Statement view lines begin at line 1.

Number of lines

INPUT; BINARY(4)

The number of lines of the statement view to be retrieved. This number includes the line specified in the start line number parameter. If fewer lines than requested are available, the number of lines placed in the receiver variable may be less than the number specified. No more than the number of lines specified is placed in the receiver variable.

The following special value is supported for this parameter:

0	All lines from the start line number to the end of the statement view should be retrieved.

Error code

I/O; CHAR(*)

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

Format of Receiver Variable

The receiver variable consists of:

- A receiver variable header section.
- A section containing the statement view lines.
- A section containing information for each procedure in the statement view.
- A string space containing the statement view procedure names.
- A section containing offsets to additional information about individual statement view lines.
- A section containing additional information about individual statement view lines.
- A space containing variable length fields that are referenced by other returned data sections.

Variables containing offsets are used to access statement view data. All offsets are relative to the beginning of the receiver variable.

Receiver Variable Header Section

The following table shows the information supplied in the receiver variable parameter. For more information on each field, see Field Descriptions.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Offset to first statement view line
12	C	BINARY(4)	Number of lines returned
16	10	BINARY(4)	Length of statement view line
20	14	BINARY(4)	Offset to first procedure information structure
24	18	BINARY(4)	Offset to first statement-view-line additional-information offset.

Statement View Section

The statement view is returned as an array of statement lines. The first statement view line can be accessed by using the first view line offset in the receiver header. The number of lines returned variable in the receiver header is used to tell how many statement lines were returned. The total number of bytes in each line is equal to the line length. Each line has the following format.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Statement number
4	4	BINARY(4)	Statement type
8	8	BINARY(4)	Offset to statement procedure information structure

Procedure Information Section

The procedure information section contains one variable-length data structure for each procedure in the statement view. The first procedure information data structure can be accessed by using the first procedure information offset in the receiver header. Each statement view line contains a statement procedure information offset that can be used to locate procedure information for the statement line. Each procedure information data structure has the following format.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Offset to next procedure information structure
4	4	BINARY(4)	Procedure dictionary number
8	8	BINARY(4)	Offset to procedure name
12	C	BINARY(4)	Length of procedure name

16	10	BINARY(4)	Offset to first statement line range element
20	14	BINARY(4)	Number of statement line ranges
Note: The following fields are repeated for each statement line range.			
		BINARY(4)	Low line number
		BINARY(4)	High line number

Procedure Name String Space

The procedure name string space contains the text of the procedure names in the module. The procedure name offset in the procedure information section is used to access a procedure name. The procedure name length is also contained in the procedure information section. The procedure name is converted to the coded character set identifier (CCSID) of the job.

Offset			
Dec	**Hex**	**Type**	**Field**
		CHAR(*)	Procedure name

Statement-View-Line Additional-Information Offsets Section

If the compiler supplies it, additional information is returned for individual statement view lines. For example, a statement may have a name associated with it, such as a block or label name. Each line in the statement view section has a corresponding offset to additional information for the line. Thus, the first offset in this section is used to find the additional information for the first statement view line returned. The second offset will reference additional information for the second statement view line returned, and so on. There must be space in the receiver variable for the additional-information offsets of all statement view lines returned or none of the offsets is returned. The presence of this section is indicated by a nonzero value in the offset to first statement-view-line additional-information offset in the receiver header. If this section is present, there is one offset for each statement view line returned. If there is additional information for a statement view line, the additional information offset for it is nonzero. Each offset has the following format.

Offset			
Dec	**Hex**	**Type**	**Field**
Note: The following field is repeated for each statement view line returned.			
0	0	BINARY(4)	Offset to statement-view-line additional information

Statement-View-Line Additional-Information Section

If the compiler supplies it, additional information is returned for individual statement view lines. For example, a statement may have a name associated with it, such as a block or label name. The statement-view-line additional-information section contains one variable-length data structure for each statement view line that has additional information associated with it. If there is not enough room in the receiver variable for all of the additional-information data structures to be returned, the number that fits is returned. The additional information data structures are referenced by the offsets in the statement-view-line additional-information offsets section. Each additional-information data structure has the following format.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Offset to statement name
4	4	BINARY(4)	Length of statement name

Variable Length Field Section

This section contains space to return variable length fields. These fields are referenced by other returned data structures through offsets. Usually, a length field would also be contained within the same data structure that references a field in this space.

Offset			
Dec	Hex	Type	Field
		CHAR(*)	Variable length field

Field Descriptions

Bytes available. The number of bytes of data available to be returned. All available data is returned if enough space is provided.

Bytes returned. The number of bytes of data returned.

High line number. The high view-line number in the statement view of a procedure statement range.

Length of procedure name. The length of the procedure name in the procedure string space. For OPM programs the procedure name length is set to a value of 1.

Length of statement name. The length of the statement name associated with the statement view line.

Length of statement view line. The length of each statement view line in the statement view section.

Low line number. The low view-line number in the statement view of a procedure statement range.

Number of lines returned. The number of statement view lines retrieved by this API. This may be less than the number of lines requested or available if the receiver variable is not large enough to hold the number of lines requested.

Number of statement line ranges. The number of statement view line ranges in the procedure information data structure.

Offset to first procedure information structure. The displacement from the start of the receiver variable to the first procedure information data structure in the procedure information section. This value is zero when no procedure information is returned because of insufficient receiver variable space.

Offset to first statement line range element. The displacement from the start of the receiver variable to the first statement range element in the procedure information data structure.

Offset to first statement view line. The displacement from the start of the receiver variable to the first statement view line. This value is zero if no statement view lines are returned because of insufficient receiver variable space.

Offset to first statement-view-line additional-information offset. The displacement from the start of the receiver variable to the first statement-view-line additional-information offset. This value is zero if no statement-view-line additional-information offsets are returned because of insufficient receiver variable space, or if the compiler does not support debug data for additional statement view lines.

Offset to next procedure information structure. The displacement from the start of the receiver variable to the next procedure information data structure. This value is zero when there are no more procedure information data structures.

Offset to procedure name. The displacement from the start of the receiver variable to the procedure name. This value is zero if the procedure name is not returned because it would not fit in the procedure string space.

Offset to statement name. The displacement from the start of the receiver variable to the statement name that is associated with the statement view line. For example, this could be a block or label name. This value is zero if the statement name is not returned because it would not fit in the variable length field section, or because the compiler did not provide a statement name.

Offset to statement procedure information structure. The displacement from the start of the receiver variable to appropriate procedure information data structure in the procedure information section. This value is zero if the procedure information for this statement was not returned because of insufficient receiver-variable space.

Offset to statement-view-line additional information. The displacement from the start of the receiver variable to the statement-view-line additional-information data structure. This value is zero if no statement-view-line additional information is returned because of insufficient receiver variable space, or because there is no additional information for the statement view line.

Procedure dictionary number. The number that uniquely identifies the procedure in this module. For OPM programs the procedure dictionary number is set to a value of 0.

Procedure name. The name of the procedure. The procedure name is converted to the CCSID of the job. For OPM programs the procedure name is set to a blank value with a length of 1 byte.

Statement number. The number that uniquely identifies the statement in the procedure. This number is shown on the compiler listing. For OPM programs the statement number is the same as the machine interface (MI) number.

Statement type. The type number of statement produced by the compiler. Possible values are as follows:

1	INIT CODE
2	PROC ENTRY
3	PROC EXIT
4	ALLOC
5	STMT
6	ENTRY
7	EXIT
8	MULTIEXIT
9	PATH LABEL

10	PATH CALL BGN
11	PATH CALL RET
12	PATH DO BGN
13	PATH TRUEIF
14	PATH FALSEIF
15	PATH WHEN BGN
16	PATH OTHERW
17	GOTO
18	POST COMPOUND

Variable length field. A field referenced by an offset in a returned data structure. The data type is determined by where it is referenced. For example, a statement name field is a text string.

Error Messages

Message ID	**Error Message Text**
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF954A E	No source text available.
CPF9563 E	Number of lines not valid.
CPF9564 E	Starting line number not valid.
CPF9582 E	View is not a statement view.

API Introduced: V3R6

Retrieve Stopped Position (QteRetrieveStoppedPosition) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	View ID	Input	Binary(4)
4	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Retrieve Stopped Position (QteRetrieveStoppedPosition) API is used to determine if a module in a program is on the call stack. It indicates the position in the view at which the program stopped if the program is on the stack. The caller must specify a registered view ID. The most recently called procedure in the specified module is the one whose line is returned. If a program is on the stack, the stack is searched from the most recent call backward until a procedure in the module is found. The location in that procedure is returned.

If no procedure in the identified module is on the stack, a zero is returned.

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The variable that is to receive the information requested. You can specify the size of this area to be smaller than the format requested if you specify the length of receiver variable parameter correctly. As a result, the API returns only the data that the area can hold. For more information, see Format of Receiver Variable.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. The minimum length is 8 bytes.

It is suggested that a receiver variable length be given that is large enough to hold one position element. Because this normally is the number of elements that are returned, a single call to this API is usually sufficient. Also, this allows the number of stopped positions field to be used to determine whether the program is stopped. If zero elements are returned, the program is not stopped in the specified view.

View ID

INPUT; BINARY(4)

The identifier of a previously registered view obtained using the Register Debug View API.

Error code

I/O; CHAR(*)

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

Format of Receiver Variable

The following table shows the information supplied in the receiver variable parameter. For more information on the fields see, Field Descriptions.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of stopped positions
Note: The following fields are repeated for each stopped position.			
		BINARY(4)	Line number
		BINARY(4)	Column number

Field Descriptions

Bytes available. The number of bytes of data available to be returned. All available data is returned if enough space is provided.

Bytes returned. The number of bytes of data returned.

Column number. The column number within the line number specified where the program is stopped in the view ID. Column numbers can be 1 through 255.

Line number. The line number within the view ID where the program is stopped. This number represents the line number within the view that corresponds to text retrieved using the Retrieve View Text API.

Number of stopped positions. A stopped position consists of the line number and column number fields and are repeated this number of times, once for each position available. If the view is not on the stack, this number is zero. If there is no room in the receiver variable to hold any stopped positions, this number is also zero. Therefore, there should be enough room in the receiver variable to hold at least one stopped position.

Because of program optimization, it is possible for the API not to know exactly where the view is stopped. For this reason, more than one position may be returned.

Error Messages

Message ID	Error Message Text
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9548 E	Map not available.
CPF9549 E	Error addressing API parameter.

API Introduced: V2R3

Top | Debugger APIs | APIs by category

Retrieve View File (QteRetrieveViewFile) API

Required Parameter Group:

1	Text descriptor receiver variable	Output	Char(*)
2	Length of text descriptor receiver variable	Input	Binary(4)
3	File name receiver variable	Output	Char(*)
4	Length of file name receiver variable	Input	Binary(4)
5	Format of file name receiver variable	Input	Char(8)
6	View ID	Input	Binary(4)
7	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Retrieve View File (QteRetrieveViewFile) API is used to retrieve all the files and text information necessary to construct the text for the entire view specified by the view ID parameter. A list of text descriptors is returned. Each text descriptor describes where a piece of text for the view comes from, either from a file specified in the file name receiver variable or from supplied text that may be obtained using the Retrieve View Text API.

Required Parameter Group

Text descriptor receiver variable

OUTPUT; CHAR(*)

The output variable containing the list of text descriptors, which describe how the specified view is constructed. For more information, see Format of Text Descriptor Receiver Variable.

Length of text descriptor receiver variable

INPUT; BINARY(4)

The length in bytes of the text descriptor receiver variable parameter. The minimum length is 8 bytes.

File name receiver variable

OUTPUT; CHAR(*)

The output variable containing the list of files referenced by the text descriptor receiver variable.

Length of file name receiver variable

INPUT; BINARY(4)

The length in bytes of the file name receiver variable. The minimum length is 8 bytes.

Format of file name receiver variable

INPUT; CHAR(8)

The content and format of the information to be supplied by the API in the file name receiver variable. The only valid value is:

RVFN0100	Format of file name receiver variable

For more information, see Format of File Name Receiver Variable.

View ID

INPUT; BINARY(4)

The identifier of a previously registered view obtained by using the Register Debug View API.

Error code

I/O; CHAR(*)

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

Format of Text Descriptor Receiver Variable

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of text descriptor entries
Note: The following three fields are repeated the number of times specified in the number of text descriptor entries field.			
		BINARY(4)	File name index
		BINARY(4)	Line number
		BINARY(4)	Number of lines

Format of File Name Receiver Variable

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of file name entries
Note: The following eight fields are repeated the number of times specified on the number of file name entries field.			
		BINARY(4)	Offset of file name
		BINARY(4)	Length of file name
		CHAR(8)	File format name
		BINARY(4)	External or OS/400 IFS file flag

		BINARY(4)	CCSID of file name
		CHAR(2)	Country or region ID of file name
		CHAR(3)	Language ID of file name
		CHAR(3)	Reserved
Note: The file names buffer follows all file name entries.			
		CHAR(*)	File names buffer

Field Descriptions

Bytes available. The number of bytes of data available to be returned. All available data is returned if enough space is provided.

Bytes returned. The number of bytes of data returned.

CCSID of file name. The CCSID the file name is in. The value of this field is only valid for file format name RTVF0200.

Country or region ID of file name. The country or region ID of the file name. The value of this field is valid for file format name RTVF0200 only.

External or IFS file flag. Whether the file is an OS/400 integrated file system file or an external file. A value of 0 means external file; a value of 1 means OS/400 integrated file system file. The value of this field is valid only for file format name RTVF0200.

File names buffer. A list of file names from which text should be retrieved.

File name index. An index into the file name receiver variable array. 0 is the first file entry in the file name receiver variable. If the index is -1, the text comes from supplied text.

File format name. The format of a file in the file names buffer. Possible formats are:

RTVF0100	OS/400 file (see RTVF0100 Format)
RTVF0200	External or OS/400 HFS file (see RTVF0200 Format)

Language ID of file name. The language ID of the file name. The value of this field is valid only for file format name RTVF0200.

Length of file name. The length in bytes of a file name in the file names buffer.

Line number. The line number in the file that is referenced by the file name index to start reading text from. If the file name index is -1, this specifies the line number in the view where the supplied text can be retrieved using the QteRetrieveViewText API.

Number of file name entries. The number of entries returned in the file name receiver variable.

Number of lines. The number of lines of text described by the text descriptor. The number of lines to read from the file, which is the number of lines of supplied text to be retrieved using the QteRetrieveViewText API.

Number of text descriptor entries. The number of entries returned in the receiver variable. The file name index, line number, and number of lines fields are repeated this number of times.

Offset of file name. From the start of the file names buffer, the start of a file name.

Formats of File Format Name

RTVF0100 Format

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(10)	OS/400 file name
10	A	CHAR(10)	OS/400 library
20	14	CHAR(10)	OS/400 member name

Field Descriptions

OS/400 file name. The name of an OS/400 file from which text should be retrieved. It is an OS/400 object name, left-justified, and padded with blanks.

OS/400 library. The name of a library that contains the file from which text should be retrieved. It is an OS/400 object name, left-justified, and padded with blanks.

OS/400 member name. The name of the member of the file from which text should be retrieved. It is an OS/400 object name, left-justified, and padded with blanks.

RTVF0200 Format

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(*)	External file or OS/400 integrated file system file name

Field Description

External file or OS/400 integrated file system file name. The name of an external file or OS/400 integrated file system file from which text should be retrieved. The value of this field is valid only for file format name RTVF0200.

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.

CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF954A E	No source text available.

API Introduced: V3R1

Retrieve View Line Information (QteRetrieveViewLineInformation) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Receiver variable length	Input	Binary(4)
3	Format name	Input	Char(8)
4	View ID	Input	Binary(4)
5	Start line number	Input	Binary(4)
6	Number of lines	Input	Binary(4)
7	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Retrieve View Line Information (QteRetrieveViewLineInformation) API is used to retrieve information about a specified number of lines in a registered view.

The data returned to the caller of the API indicates whether a given line or range of lines within a view can be run or not.

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The receiver variable that receives the information requested. You can specify the size of the area to be smaller than the format requested as long as you specify the length parameter correctly. As a result, the API returns only the data that the area can hold.

See RTVL0100 Format for details on the format of the receiver variable.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable provided by the receiver variable parameter. If this value is larger than the actual amount of storage allocated for the receiver variable, the results are not predictable. The minimum length is 8 bytes.

Format name

INPUT; CHAR(8)

The format of the information returned. The possible format names are:

RTVL0100	Retrieve view line information.

View ID

INPUT; BINARY(4)

The identifier of a previously registered view obtained by using the Register Debug View (QteRegisterDebugView) API.

Start line number

INPUT; BINARY(4)

The number of the first line in the view for which the API is to retrieve information. This must be greater than or equal to 1 and less than or equal to the total number of lines in the view.

Number of lines

INPUT; BINARY(4)

The number of lines in the view for which the API is to retrieve information. This number includes the line specified in the start line number parameter. Fewer than number of lines elements may be placed in the receiver variable if fewer lines than requested are available. No more than number of lines elements are placed in the receiver variable.

The following special values are supported for this parameter:

-1	All lines associated with this view starting at the value specified for the start line number parameter should be processed.

Error code

I/O; CHAR(*)

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

RTVL0100 Format

For a description of the fields in the receiver variable, see Field Descriptions.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Offset to line information array
12	C	BINARY(4)	Number of line information array elements
16	10	BINARY(4)	Length of line information array element
		CHAR(*)	Reserved
Note: The following fields describe an element in the line information array and are repeated the "number of line information array elements" times. The nth element of the array (n > 0) describes the start line number + n-1 line in the view, where start line number is a parameter to this API.			
		CHAR(1)	Line disposition

		CHAR(*)	Reserved

Field Descriptions

Bytes available. The number of bytes of data available to be returned. All available data is returned if enough space is provided.

Bytes returned. The number of bytes of data returned.

Length of line information array element. The number of bytes occupied by a single element of the line information array. Line information array elements are contiguous and all have the same length.

Line disposition. Whether the line in the view described by this array element can be run or not. Possible values are:

0	Line cannot be run
1	Line can be run

Number of line information array elements. The number of elements in the line information array that were returned by this API.

Offset to line information array. The offset (in bytes) from the start of the receiver variable to the first element of the line information array.

Reserved. An ignored field.

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF9564 E	Starting line number not valid.
CPF957A E	Number of lines not valid.
CPF957B E	Required information not found for operation.

API Introduced: V3R6

Retrieve View Text (QteRetrieveViewText) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	View ID	Input	Binary(4)
4	Start line number	Input	Binary(4)
5	Number of lines	Input	Binary(4)
6	Line length	Input	Binary(4)
7	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The Retrieve View Text (QteRetrieveViewText) API is used to retrieve source text from a specified view. This text may be formatted and displayed by the user of this API. The caller must specify the following:

- A registered view ID
- The starting line number to be retrieved
- The number of lines of text to retrieve
- A buffer to contain the text retrieved

All text retrieved, whether it comes from files or as text supplied by a processor, is in the CCSID of the job.

If source files have changed since the view was created, diagnostic messages CPF9561 (for OS/400 files) and CPF9596 (for OS/400 integrated file system files) are sent to the calling program's message queue for each file. Error messages CPF9566 (for OS/400 files) and CPF9597 (for OS/400 integrated file system files) also are issued, and all of the text available is retrieved. The calling program should warn the user that the view text may be incorrect.

If a source file cannot be accessed because it is deleted or the user is not authorized, error messages CPF9565 (for OS/400 files) and CPF9598 (for OS/400 integrated file system files) are issued. No more text is retrieved. Text up to that file is retrieved and this is indicated in the fields of the receiver variable. If the calling program attempts to read text in the view following the file, the starting line number can be set to a line after the file. The number of lines in the file that should have been read is returned in the exception data. This allows the calling program to skip over this file if desired.

It is suggested that the calling program buffer the retrieved text to minimize use of this API. Source files accessed by this API do not remain open across API calls. Performance degradation occurs for every use of the API that results in file access because of opening and closing files.

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The variable that is to receive the information requested. You can specify the size of this area to be smaller than the format requested if you specify the length of receiver variable parameter correctly. As a result, the API returns only the data that the area can hold. For more information, see Format of Receiver Variable.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable parameter. The minimum length is 8 bytes.

View ID

INPUT; BINARY(4)

The identifier of a previously registered view obtained by using the Register Debug View API.

Start line number

INPUT; BINARY(4)

The number of the first line to be retrieved.

Number of lines

INPUT; BINARY(4)

The number of lines of source text to be retrieved. This number includes the line specified in the start line number parameter. Fewer than the number of lines may be placed in the receiver variable if fewer lines than requested are available. No more than the number of lines specified is placed in the receiver variable.

The following special value is supported for this parameter:

0	All of the text associated with this view should be retrieved.

Line length

INPUT; BINARY(4)

The length of each line of text to be retrieved. Each line takes exactly this many characters. If the actual line of text is shorter, it is padded to the right with blanks. If the line is longer than this length, it is truncated to fit. The line length must be a number from 1 through 255.

Error code

I/O; CHAR(*)

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

Format of Receiver Variable

The following tables show the information supplied in the receiver variable parameter. The information returned depends on the type of view being used. For more information on each field, see Field Descriptions. For the listing view:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of lines returned
12	C	BINARY(4)	Line length
Note: The following field is repeated for each line returned. The number of characters is equal to the line length.			
		CHAR(*)	Listing view source line

For the statement view:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of lines returned
12	C	BINARY(4)	Line length
Note: The following fields are repeated for each line returned. The total number of characters in each line is equal to the line length (10 + 10 + 10 + * = line length).			
		CHAR(10)	Procedure dictionary number
		CHAR(10)	Statement number
		CHAR(10)	Statement type number
		CHAR(*)	Procedure name

For the text view:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of lines returned
12	C	BINARY(4)	Line length
Note: The following fields are repeated for each line returned.			
		CHAR(12)	Sequence number
		CHAR(*)	Text view source line

Field Descriptions

Bytes available. The number of bytes of data available to be returned.

Bytes returned. The number of bytes of data returned.

Line length. The length of each line of text in the receiver variable parameter.

Listing view source line. The text associated with each line retrieved. The number of characters in each line is equal to the line length.

Number of lines returned. The number of lines of source text retrieved by this API and available in the receiver variable. This may be less than the number of lines requested or available, if the receiver variable is not large enough to hold the text requested.

Procedure dictionary number. The number that uniquely identifies the procedure in this module. The number is left-justified and padded on the right with blanks. For OPM programs the procedure dictionary number is set to a value of 0.

Procedure name. The name of the procedure. The name is left-justified and padded on the right with blanks. For OPM programs the procedure name is blanks.

Sequence number. If the text is from a source physical file, these 12 bytes contain the sequence number and source date for that line. If the text is from an OS/400 integrated file system file, these 12 bytes are blank. If the text is supplied by the compiler, these 12 bytes are blank.

Statement number. The number that uniquely identifies the statement in the procedure. The number is left-justified and padded on the right with blanks. This number is shown on the compiler listing. For OPM programs the statement number is the same as the machine interface (MI) number.

Statement type number. The type of statement produced by the compiler. The number is left-justified and padded on the right with blanks. Possible values are:

1	INIT CODE
2	PROC ENTRY
3	PROC EXIT
4	ALLOC
5	STMT
6	ENTRY
7	EXIT
8	MULTIEXIT
9	PATH LABEL
10	PATH CALL BGN
11	PATH CALL RET
12	PATH DO BGN
13	PATH TRUEIF
14	PATH FALSEIF
15	PATH WHEN BGN
16	PATH OTHERW

17	GOTO
18	POST COMPOUND

Text view source line. The text associated with each line retrieved. The number of characters in each line equals the line length minus 12 bytes (the sequence number).

Error Messages

Message ID	Error Message Text
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF954A E	No source text available.
CPF954C E	Cannot retrieve text from file.
CPF9560 E	Line length not valid.
CPF9561 E	Source file has changed.
CPF9563 E	Number of lines not valid.
CPF9564 E	Starting line number not valid.
CPF9565 E	Source cannot be accessed.
CPF9566 E	One or more source files have changed.
CPF9596 E	Source file has changed.
CPF9597 E	One or more source files have changed.
CPF9598 E	Source file cannot be accessed.
CPF959A E	Source file type not valid.

API Introduced: V2R3

Top | Debugger APIs | APIs by category

Fast-path Debugger APIs

Fast-path debugger APIs allow the caller to bypass the generalized Debug Command API for some of the simpler, but more common, source debugging functions.

The fast-path debugger API are:

- Add Breakpoint (QteAddBreakpoint) adds a breakpoint to the specified location in a registered view.
- Remove All Breakpoints (QteRemoveAllBreakpoints) removes all breakpoints from all modules in a program.
- Remove Breakpoint (QteRemoveBreakpoint) removes a breakpoint from the specified location in a registered view.
- Step (QteStep) adds a step to a program specifying that the program will run one or more statements after which program processing is suspended.

Add Breakpoint (QteAddBreakpoint) API

Required Parameter Group:

1	View ID	Input	Binary(4)
2	Line number	Input	Binary(4)
3	Column number	Input	Binary(4)
4	Line in statement view	Output	Binary(4)
5	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTEDBGS

Threadsafe: No

The calling program uses the Add Breakpoint (QteAddBreakpoint) API to add a breakpoint at a location in a registered view.

Required Parameter Group

View ID

INPUT; BINARY(4)

The identifier of a previously registered view obtained by using the Register Debug View API.

Line number

INPUT; BINARY(4)

The line in the *View ID* where the breakpoint is to be added.

Column number

INPUT; BINARY(4)

The column in the line where the breakpoint is to be added.

Note: At this time, column numbers are ignored. Column one must be specified.

Line in statement view

OUTPUT; BINARY(4)

The API returns the line number in the statement view where the breakpoint was added.

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
CPF1938 E	Command is not allowed while serviced job is not active.
CPF1939 E	Time-out occurred waiting for a reply from the serviced job.
CPF1941 E	Serviced job has completed. Debug commands are not allowed.
CPF3CF1 E	Error code parameter not valid.
CPF7102 E	Unable to add breakpoint or trace.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF9567 E	Column number not valid.
CPF9568 E	Line number not valid.

API Introduced: V3R1

Remove All Breakpoints (QteRemoveAllBreakpoints) API

Required Parameter Group:			
1	View ID	Input	Binary(4)
2	Remove type	Input	Char(10)
3	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The calling program uses the Remove All Breakpoints (QteRemoveAllBreakpoints) API to remove all breakpoints from a program. All breakpoints in all modules will be removed, even though only one view in the program is specified. It does not matter which view of the program is specified, as long as it is a registered view.

Required Parameter Group

View ID

INPUT; BINARY(4)

The identifier of a previously registered view obtained by using the Register Debug View API.

Remove type

INPUT; CHAR(10)

Specifies which breakpoints are to be removed. The following is allowed:

**PGM*	All breakpoints in the program or service program specified by view ID are removed.

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
CPF1938 E	Command is not allowed while serviced job is not active.
CPF1939 E	Time-out occurred waiting for a reply from the serviced job.
CPF1941 E	Serviced job has completed. Debug commands are not allowed.
CPF3CF1 E	Error code parameter not valid.
CPF9541 E	Not in debug mode
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF9578 E	Remove type not valid.

API Introcuced: V3R1

Remove Breakpoint (QteRemoveBreakpoint) API

Required Parameter Group:

1	View ID	Input	Binary(4)
2	Line number	Input	Binary(4)
3	Column number	Input	Binary(4)
4	Line in statement view	Output	Binary(4)
5	Error code	I/O	Char(*)

Service Program: QTEDBGS

Threadsafe: No

The calling program uses the Remove Breakpoint (QteRemoveBreakpoint) API to remove a breakpoint from a location in a registered view. The API will complete normally whether or not there was actually a breakpoint previously added to that location.

Required Parameter Group

View ID

INPUT; BINARY(4)

The identifier of a previously registered view obtained by using the Register Debug View API.

Line number

INPUT; BINARY(4)

The line in the view ID where the breakpoint is to be removed.

Column number

INPUT; BINARY(4)

The column in the line where the breakpoint is to be removed.

Note: At this time, column numbers are ignored. Column one must be specified.

Line in statement view

OUTPUT; BINARY(4)

The API returns the line number in the statement view where the breakpoint was removed.

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
CPF1938 E	Command is not allowed while serviced job is not active.
CPF1939 E	Time-out occurred waiting for a reply from the serviced job.
CPF1941 E	Serviced job has completed. Debug commands are not allowed.
CPF3CF1 E	Error code parameter not valid.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF9567 E	Column number not valid.
CPF9568 E	Line number not valid.

API Introduced: V3R1

Top | Debugger APIs | APIs by category

Step (QteStep) API

Required Parameter Group:

1	View ID	Input	Binary(4)
2	Step count	Input	Binary(4)
3	Step type	Input	Char(10)
4	Error code	I/O	Char(*)

Default Public Authority: *USE

Service Program: QTEDBGS

Threadsafe: No

The calling program uses the Step (QteStep) API to start a step in a program. A step count is specified. When the number of statements specified by the step count is run, the program will be stopped.

Required Parameter Group

View ID

INPUT; BINARY(4)

The identifier of a previously registered view obtained by using the Register Debug View API.

Step count

INPUT; BINARY(4)

The number of statements to be run before the program is to be stopped.

Step type

INPUT; CHAR(10)

Which statements are counted when stepping in the program. The following are allowed:

**INTO*	Statements in the procedure currently stopped in are counted. Also, if that procedure calls other procedures, these statements are also counted as they are run. Thus, it is possible to stop the program in a procedure called by the procedure currently stopped.
**OVER*	Only statements in the procedure currently stopped in are counted in the step. Thus, procedures that this procedure calls are stepped over when doing the step. If the program is not currently stopped, then the step count will start with the first procedure called in that program, and all procedures that are called by this procedure are not stepped into, and their statements are not counted.

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
CPF1938 E	Command is not allowed while serviced job is not active.
CPF1939 E	Time-out occurred waiting for a reply from the serviced job.
CPF1941 E	Serviced job has completed. Debug commands are not allowed.
CPF3CF1 E	Error code parameter not valid.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.
CPF9576 E	Step count not valid.
CPF9577 E	Step type not valid.

API Introduced: V3R1

Submit Debug Command (QteSubmitDebugCommand) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	View ID	Input	Binary(4)
4	Input buffer	Input	Char(*)
5	Input buffer length	Input	Binary(4)
6	Compiler ID	Input	Char(20)
7	Error Code	I/O	Char(*)

Service Program Name: QTEDBGS

Default Public Authority: *USE

Threadsafe: No

The Submit Debug Command (QteSubmitDebugCommand) API allows a client program to issue debug language statements. Debug language statements permit client programs to enter breakpoints, run one or more statements of the program under investigation (step), and evaluate expressions. Watch conditions may also be entered to cause a breakpoint when the contents at a specified storage location are changed.

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The variable that is to receive the results of the Submit Debug Command API. For more information on the structure of the receiver variable, see Variations in Receiver Variable Structure.

The Submit Debug Command API may have more data to return than can be stored in the receiver variable. The bytes available field, described in Variations in Receiver Variable Structure, specifies how large the receiver variable must be to contain the results for the Debug Language statements submitted. If more data is available than the receiver variable can contain, a larger buffer should be provided and the API should be reissued.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. If the length is larger than the size of the receiver variable, the results may not be predictable. The minimum length is 8 bytes.

View ID

INPUT; BINARY(4)

An identifier of a view of a module whose operation is managed by the source debugger. The view

ID is returned as a result of issuing the Register Debug View API. The view ID is used to find debug data associated with the module.

Input buffer

INPUT; CHAR(*)

The input variable that is passed to the Submit Debug Command API. The information passed in the buffer is debug language statements.

Input buffer length

INPUT; BINARY(4)

The length of the data provided in the input buffer.

Compiler ID

INPUT; CHAR(20)

The compiler ID of the compiler that produced the module being debugged. This information is used by the debug translator during expression evaluation. The compiler ID is returned by the Retrieve Module Views (QteRetrieveModuleViews) API.

Error code

I/O; CHAR(*)

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

Receiver Variable Format

The following table shows the structure of the receiver variable. For more information on the fields contained in the table, see Field Descriptions.

Receiver Variable Structure

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Entry count
12	C	CHAR(*)	Results array
		CHAR(*)	String space

Field Descriptions

Bytes available. The number of bytes of data available to be returned. All available data is returned if enough space is provided.

Bytes returned. The number of bytes of data returned.

Entry count. The number of entries in the results array. The value of the field is the number of entries in the results array. Each entry occupies 12 bytes. Depending on the kind of information returned, values in entries vary.

Results array. The results of interpreting debug language statements. This is an array of records having similar structures. Each record in the array occupies 12 bytes. There can be up to three fields in each record. Each field occupies 4 bytes and can be interpreted as an unsigned (nonnegative) integer. The first field in a record is the result type field and is used to select the remaining fields. Entries in the result record array fall into several classes. Variations in Receiver Variable Structure depicts several formats of result records.

Statements are interpreted sequentially and the results of each statement are placed in the order in which statements appear in the input buffer. The evaluate statement can return many values if an array or a structure is evaluated. The entry count field contains the number of entries in the results array, and the structure of each entry is summarized in Variations in Receiver Variable Structure.

String space. A sequence of strings. Each string is an array of characters whose last character is a null character.

Description of the Structure of the Receiver Variable

Variations in Receiver Variable Structure illustrates three possible variations in the structure of the receiver variable. The receiver variable consists of the following structures:

- A header record
  This structure consists of three fields:
  - Bytes returned
  - Bytes available
  - Entry count
- A result array
- A string space

Variations in Receiver Variable Structure

Header	Bytes Returned	Bytes Available	Entry Count
Result array 1	Result type		
Result array 2	Result type	Count	
Result array 3	Result type	Offset	Length
String space			

Each row of Variations in Receiver Variable Structure occupies 12 bytes. The row containing the headings describes the remainder of the receiver variable. The number of bytes returned is assigned to that field. The value of the bytes returned field is always less than or equal to the size of the receiver variable. The number of bytes available may be greater than the number returned. In that case, the client program should reissue the Submit Debug Command API to obtain all data produced for the input debug language statements. The entry count field in the first row indicates the number of 12-byte records, each beginning with a result type field, that follow.

Records beginning with a result type field have the following basic formats.

- The first entry in the array shows a record containing only one field, result type. Records having this structure acknowledge that a kind of debug language statement was translated. An example of this kind of record is the result record for a CLEAR PGM statement.

- The second entry in the array shows a record containing a count field as well as a result type field. The count field can serve two purposes:

    - It can acknowledge that a debug language statement was properly translated as in the case of the StepR result record.

    - It can enumerate the number of related records to follow as in the case of the BreakR result record.

- The offset field contains the displacement from the start of the receiver variable to the first byte of the character string. All character strings are stored at the end of the receiver variable directly after the record entries. Displacements are measured in bytes.

  The length field contains the number of characters in the character string.

  The last character of each string in the string space has an ordinal value of zero. All characters in the string space occupy 8 bits. The length of a string in the string space does not include the last character.

  > **Note:** The length field will always be set to 512, with each of the 512 characters occupying 16 bits, for a string of type kStringF_E when debugging a JAVA executable. This will occur even when the returned string has a length of less than 512. The end of the returned string can be found by locating the first unicode character in the string that has an ordinal value of zero. As unicode, this character will occupy 16 bits.

- The last row of Variations in Receiver Variable Structure depicts an arbitrarily large string space containing character data. Names and other text fields that are referred to in the result type fields shown in the other rows of Variations in Receiver Variable Structure are stored in this area.

Results Array Entry Structure Summary

The following tables describe each result record in detail. Each result record contains up to three fields and always occupies 12 bytes. The first field, the result type field, is used as an enumerated type. The result type field determines the format of each result record.

Each of the following enumeration constants has both a symbolic name and an ordinal value. The terms symbolic and ordinal refer to enumerations found in programming languages. The symbolic value of an enumeration constant is the symbol, usually a descriptive word that serves as a keyword for the programmer (for example, StepR). The ordinal value of an enumeration constant is the integer constant assigned, usually by the compiler, to the symbolic value. For example, 1 is assigned for StepR.

StepR (1)

Record format StepR is returned as a result of evaluating a step statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Step count
8	8	CHAR(4)	Reserved

BreakR (2)

Record format BreakR contains the number of records returned for a break statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Break results count
8	8	CHAR(4)	Reserved

ClearBreakpointR (3)

Record format ClearBreakpointR contains the line number of the breakpoint removed as a result of interpreting the CLEAR break-position statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Line number
8	8	CHAR(4)	Reserved

ClearPgmR (4)

Record format ClearPgmR indicates that all breakpoints have been removed in the current program as result of interpreting the CLEAR PGM statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	CHAR(4)	Reserved
8	8	CHAR(4)	Reserved

BreakPositionR (5)

Record format BreakPositionR identifies the line number on which a breakpoint was entered. This may not be the same line number as the one entered in the break statement.

Offset			

Dec	Hex	Type	Field
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Line number
8	8	CHAR(4)	Reserved

EvaluationR (6)

Record format EvaluationR contains the number of records returned for an evaluate statement that are referred to in the subsequent ExpressionValueR record.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Evaluation count
8	8	BINARY(4)	Reserved

ExpressionTextR (7)

Record format ExpressionTextR describes a character string that contains the expression that was evaluated by the evaluate statement.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Expression text offset
8	8	BINARY(4)	Expression text length

ExpressionValueR (8)

Record format ExpressionValueR refers to text that contains the formatted value of the expression that is described by the ExpressionTextR record.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Expression value offset
8	8	BINARY(4)	Expression value length

ExpressionTypeR (9)

Record format ExpressionTypeR contains the type of the expression whose value is referred to in the ExpressionValueR record.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Expression type
8	8	CHAR(4)	Reserved

QualifyR (10)

Record format QualifyR is returned as a result of evaluating a qualify statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Line number
8	8	BINARY(4)	Reserved

TypeR (11)

Record format TypeR contains the number of records that are returned for an ATTR statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Type record count
8	8	BINARY(4)	Reserved

TypeDescR (12)

Record format TypeDescR contains the type and length of the program variable.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Type
8	8	BINARY(4)	Length

DecimalR (13)

Record format DecimalR is returned only for decimal type variables and contains the total and fractional number of digits in the decimal number.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Total digits
8	8	BINARY(4)	Fraction digits

ArrayR (14)

Record format ArrayR is returned only for array type variables and contains the number of dimensions in the array. The ArrayR record is followed by a DimensionR record for each dimension.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Dimensions
8	8	BINARY(4)	Reserved

DimensionR (15)

Record format DimensionR is returned only for array type variables and contains the low and high bounds of the array dimensions. There is one DimensionR record for each dimension in the array.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Low bound
8	8	BINARY(4)	High bound

WatchR (16)

Record format WatchR contains the number of records returned for a watch statement.

Offset			
Dec	Hex	Type	Field

0	0	BINARY(4)	Result type
4	4	BINARY(4)	Watch results count
8	8	BINARY(4)	Reserved

WatchNumberR (17)

Record format WatchNumberR describes the watch condition that was set as a result of the watch statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Watch number
8	8	BINARY(4)	Watch length

ClearWatchNumberR (18)

Record format ClearWatchNumberR contains the watch number that is cleared as a result of interpreting the CLEAR WATCH watch-number statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Watch number
8	8	BINARY(4)	Reserved

ClearWatchR (19)

Record format ClearWatchR indicates that all watches in this debug session have been removed as a result of interpreting the CLEAR WATCH ALL statement.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Result type
4	4	BINARY(4)	Reserved
8	8	BINARY(4)	Reserved

TBreakR (20)

Record format TBreakR contains the number of records that are returned for a tbreak statement.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Result type
4	4	BINARY(4)	TBreak results count
8	8	CHAR(4)	Reserved

SBreakR (21)

»Record format SBreakR contains the number of records that are returned for a sbreak statement.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Result type
4	4	BINARY(4)	SBreak results count
8	8	CHAR(4)	Reserved«

Field Descriptions

Break results count. The number of entries returned for the break statement.

Dimensions. The number of dimensions in the array.

Expression text length. The number of characters in the expression text.

Expression text offset. The displacement from the start of the receiver variable to the first character of the expression text. Displacement is measured in bytes.

Expression value length. The number of characters in the expression value text.

Expression value offset. The displacement from the start of the receiver variable to the first byte of the expression value text. Displacement is measured in bytes.

Evaluation count. The number of records returned for an evaluate statement.

Expression type. The data type of the expression. The expression type may be one of the following:

Type	Enumeration	Description
0	kNoType__E	Type is not valid
1	kChar__8_E	8-bit character value
2	kChar_16_E	16-bit character value
3	kBool_32_E	32-bit Boolean value
4	kCard_16_E	16-bit unsigned integer value
5	kCard_32_E	32-bit unsigned integer value

6	kInt__16_E	16-bit two's complement integer value
7	kInt__32_E	32-bit two's complement integer value
8	kReal_32_E	32-bit real floating-point value
9	kReal_64_E	64-bit real floating-point value
10	kSpcPtr__E	64 or 128-bit pointer
11	kFncPtr__E	64 or 128-bit function pointer
12	kMchAddr_E	64 or 128-bit machine pointer
13	kRecord__E	Structure or record
14	kArray___E	Array
15	kEnum____E	Enumerated type
16	kString__E	String (:s format on EVAL)
17	kPacked__E	Packed decimal
18	kZonedTE_E	Zoned, trailing embedded sign
19	kZonedTS_E	Zoned, trailing separate sign
20	kZonedLE_E	Zoned, leading embedded sign
21	kZonedLS_E	Zoned, leading separate sign
22	kBinD_16_E	16-bit binary decimal value
23	kBinD_32_E	32-bit binary decimal value
24	kBinD_64_E	64-bit binary decimal value
25	kTable___E	Multiple occurrence data structure
26	kInd_____E	Indicator
27	kDate____E	Date
28	kTime____E	Time
29	kTstamp__E	Timestamp
30	kFixedL__E	Fixed-length string
31	kStringF_E	String (:f, :a, :u format on EVAL command)
100	kHex_____E	Hexadecimal (:x format on EVAL command)

Fraction digits. The number of digits to the right of the decimal point in a decimal number.

High bound. The high boundary of the array dimension.

Length. The program variable length that is returned by the TypeDescR result record. The length units are bits.

Line number. The number of the line on which the action requested was performed.

Low bound. The low boundary of the array dimension.

Reserved. An ignored field.

Result type. The ordinal value of the result array.

SBreak results count. The number of entries returned for the sbreak statement.

Step count. The number of statements processed.

TBreak results count. The number of entries returned for the tbreak statement.

Total digits. The total number of digits in a decimal number.

Type. The program variable type that is returned by the TypeDescR result record. The meanings of this field's value are the same as the expression type field.

Type record count. The number of records returned for an attr statement.

Watch length. The length in bytes of the storage being watched for this watch condition.

Watch number. The identification number assigned to the watch condition. This number is used by various debug functions to identify individual watches.

Watch results count. The number of result records returned for the watch statement.

Statement Results

ATTR Statement Results. The Submit Debug Command API returns a description of the symbol table entry for the program variable entered. A variable number of result records may be produced:

- A TypeR record is returned, which provides a count of the number of records returned for an ATTR statement.
- A TypeDescR record is returned, which provides the type and size of the program variable.
- A DecimalR record is returned only if the program variable is a decimal type. This record describes the total and fractional digits in the decimal number.
- An ArrayR record is returned only if the program variable is an array. This record returns the number of dimensions in an array.
- A DimensionR record is returned only if the program variable is an array. This record returns the low and high bounds of the array dimensions.

Break Statement Results. The Submit Debug Command API returns a detailed description of the break-position and conditional expression of a conditional breakpoint when a break statement is translated.

The items returned follow:

- The number of records returned as a result of evaluating a break statement. Record type BreakR contains this information.
- The position on which the breakpoint was entered. Record type BreakPositionR contains the line number of the line on which the breakpoint was entered. Be aware that the input line number may be mapped to a different line. For example, a breakpoint entered on a line that contains a comment is mapped to the next line that contains an operational statement.
- The text of the expression that defines a conditional breakpoint. Record type ExpressionTextR refers to the text of the condition.

The break statement is interpreted. Program operation is managed by OS/400 according to the definition of the break statement.

Clear Statement Results. One record is returned. The record type depends on the operand following the keyword CLEAR. If the operand is a line number, the record type is ClearBreakpointR. If the operand is the keyword PGM, the record type is ClearPgmR. If the operand is WATCH and a watch number is specified, the record type is ClearWatchNumberR. If the operand is WATCH and all watches are cleared, the record type is ClearWatchR.

The ClearBreakpointR record contains the line number input for the break position.

The clear statement is interpreted. One or more breakpoints are removed from the program under investigation.

Evaluate Statement Results. An evaluate statement produces a variable number of Result Records. The first four result records follow:

- An EvaluationR record is returned, which enumerates itself and subsequent records. The EvaluationR result record always contains an evaluation count of four.
- An Expression text record is returned, which contains the offset and length of the string, which represents the expression text.
- An Expression value record is returned, which contains the offset and length of the string, which represents the value of the expression.

  Note: The length field will always be set to 512, with each of the 512 characters occupying 16 bits, for a string of type kStringF_E when debugging a JAVA executable. This will occur even when the returned string has a length of less than 512. The end of the returned string can be found by locating the first unicode character in the string that has an ordinal value of zero. As unicode, this character will occupy 16 bits.
- An Expression type record is returned, which contains the type of the expression.

A single value is returned for an arithmetic expression or scalar variable. Multiple values are returned when a structure is evaluated. Refer to Examples of Result Records Returned by Submit Debug Command API for examples of the result records returned when a structure or an array is evaluated.

The evaluate statement is interpreted. Data is formatted according to the type of the input expression. Refer to Presentation Formats for a description of presentation formats.

Qualify Statement Results. One record is returned. The value of the result type field is QualifyR. The QualifyR record contains the input line number used to establish the current locality for subsequent evaluate statements.

A reference to the block that defines the current locality is assigned by the qualify statement.

»**SBreak Statement Results.** The Submit Debug Command API returns a detailed description of the break-position when an sbreak statement is translated. The items returned are:

- The number of records returned as a result of evaluating a sbreak statement. Record type SBreakR contains this information.
- The position on which the breakpoint was entered. Record type BreakPositionR contains the line number of the line on which the breakpoint was entered. Be aware that the input line number may be mapped to a different line. For example, a breakpoint entered on a line that contains a comment is mapped to the next line that contains an operational statement.

The sbreak statement is interpreted. Program operation is managed by OS/400 according to the definition of the sbreak statement.«

Step Statement Results. One record is returned. The value of the result type field is StepR. The StepR record contains the number of statements to be run when control is given to the program under investigation.

The step statement is interpreted. Program processing is managed by OS/400 according to the definition of the step statement.

TBreak Statement Results. The Submit Debug Command API returns a detailed description of the thread break-position and conditional expression of a conditional breakpoint when a tbreak statement is translated. The items returned follow:

- The number of records returned as a result of evaluating a tbreak statement. Record type TBreakR contains this information.
- The position on which the thread breakpoint was entered. Record type BreakPositionR contains the line number of the line on which the thread breakpoint was entered. Be aware that the input line number may be mapped to a different line. For example, a thread breakpoint entered on a line that contains a comment is mapped to the next line that contains an operational statement.
- The text of the expression that defines a conditional breakpoint for a thread. Record type ExpressionTextR refers to the text of the condition.

The tbreak statement is interpreted. Program operation is managed by OS/400 according to the definition of the tbreak statement.

Watch Statement Results. The watch statement returns the following result records:

- A WatchR record is returned, which provides a count of the number of result records for the watch statement.
- A WatchNumberR record is returned, which contains the watch number assigned and the length in bytes of the storage being watched.
- An ExpressionTextR record, which contains the offset and length of a string. This record represents the watch statement expression text.
- An ExpressionValueR record, which contains the offset and length of a string. This record represents the watch storage location address. This value is always a text representation of a space pointer that contains the value of the pointer to the watched storage location (for example, 'SPP:08006F0054001004').

Examples of Result Records Returned by Submit Debug Command API

This section contains examples of result records returned by the Submit Debug Command API. Each example contains a fragment of a program, a debug language statement that appears in the input buffer, and the results produced in the receiver variable.

The null termination symbol denotes the end of a character string in the examples that follow.

Break Statement Example

C Program Fragment

Assume program operation is suspended in the program shown in Figure 1 just before line 6 runs.

Figure 1. Program for Break Example

```
Line  C Source

 1    #include  stdio.h
 2    int T[] = {1,2,3,5,7,11,13,17,23,29};
 3    int BinarySearch(int v, int f, int l);
 4    main()
 5    { int result;
 6      result = BinarySearch(17,0,9);
 7      printf("result= "); printf("%d",result); printf(" \n");
 8    }
```

Input Buffer

```
BREAK 7 WHEN result > 5
```

Receiver Variable

Offset	Field	Value
0	Bytes returned Bytes available Entry count	59 59 3
12	Result type Break results count Reserved	BreakR 3
24	Result type Line number Reserved	BreakPositionR 7
36	Result type Expression text offset Expression text length	ExpressionTextR 48 10
48	String space	result > 5

Scalar Evaluate Statement Example

C Program Fragment

Consider the C program fragment in Figure 2. Variable i defines an integer.

Figure 2. Program for Scalar Evaluate Example

```
Line  C Source

 1    int i = 29;
```

Input Buffer

```
EVAL i
```

Receiver Variable

Offset	Field	Value
0	Bytes returned Bytes available Entry count	65 65 4
12	Result type Evaluation count Reserved	EvaluationR 4
24	Result type Expression text offset Expression text length	ExpressionTextR 60 1
36	Result type Expression value offset Expression value length	ExpressionValueR 62 2
48	Result type Expression type Reserved	ExpressionTypeR kInt__32_E
60	String space	i29

Scalar Evaluate Statement Example

RPG Program Fragment

Consider the RPG program fragment in Figure 3. The fragment assigns 1 to a zoned(1,0) variable I.

The program is currently stopped at line 2.

Figure 3. RPG Programming Language Example, Evaluate

```
           CL0N01Factor1+++++++OpcodeE+Extended-factor2

1          C                   EVAL     I=1
2          C                   MOVE     *ON           *INLR
```

Input Buffer

```
EVAL I
```

Receiver Variable

Offset	Field	Value

0	Bytes returned Bytes available Entry count	64 64 4
12	Result type Evaluation count Reserved	EvaluationR 4
24	Result type Expression text offset Expression text length	ExpressionTextR 60 1
36	Result type Expression value offset Expression value length	ExpressionValueR 62 1
48	Result type Expression type Reserved	ExpressionTypeR kZonedTE_E
60	String space	I1

Structure Evaluate Statement Example

C Program Fragment

Consider the C program fragment in Figure 4.

Figure 4. Program for Structure Evaluate Example

```
C Source

struct {
    int i;
    float f;
    struct {
       char c;
       enum e {red,yellow};
    } s2;
} s1 = { 1 , 5.0, {'a' , red } };
```

Input Buffer

```
EVAL s1
```

Receiver Variable

Offset	Field	Value
0	Bytes returned Bytes available Entry count	246 246 16

12	Result type Evaluation count Reserved	EvaluationR 4
24	Result type Expression text offset Expression text length	ExpressionTextR 204 4
36	Result type Expression value offset Expression value length	ExpressionValueR 209 1
48	Result type Expression type Reserved	ExpressionTypeR kInt__32_E
60	Result type Evaluation count Reserved	EvaluationR 4
72	Result type Expression text offset Expression text length	ExpressionTextR 211 4
84	Result type Expression value offset Expression value length	ExpressionValueR 216 7
96	Result type Expression type Reserved	ExpressionTypeR kReal_64_E
108	Result type Evaluation count Reserved	EvaluationR 4
120	Result type Expression text offset Expression text length	ExpressionTextR 224 7
132	Result type Expression value offset Expression value length	ExpressionValueR 232 1
144	Result type Expression type Reserved	ExpressionTypeR kChar__8_E
156	Result type Evaluation count Reserved	EvaluationR 4
168	Result type Expression text offset Expression text length	ExpressionTextR 234 7
180	Result type Expression value offset Expression value length	ExpressionValueR 242 3
192	Result type Expression type Reserved	ExpressionTypeR kEnum____E
204	String space	See Note.
Note: s1.i1s1.f5.0E+00s1.s2.cas1.s2.ered		

Step Statement Example

C Program Fragment

Assume program operation is suspended in the program shown in Figure 5 just before line 6 runs.

Figure 5. Program for Step Example

```
Line   C Source

 1     #include  stdio.h
 2     int T[] = {1,2,3,5,7,11,13,17,23,29};
 3     int BinarySearch(int v, int f, int l);
 4     main()
 5     { int result;
 6         result = BinarySearch(17,0,9);
 7         printf("result= "); printf("%d",result); printf(" \n");
 8     }
```

Input Buffer

```
STEP
```

Receiver Variable

Offset	Field	Value
0	Bytes Returned Bytes Available Entry Count	24 24 1
12	Result type Step Count Reserved	StepR 1

ATTR Statement Example

RPG Program Fragment

Consider the RPG program fragment in Figure 6. The fragment assigns 1 to a zoned(1,0) variable I.

The program is currently stopped at line 2.

Figure 6. RPG Programming Language Example, Evaluate

```
          CL0N01Factor1+++++++OpcodeE+Extended-factor2

1         C                   EVAL    I=1
2         C                   MOVE    *ON          *INLR
```

Input Buffer

```
ATTR I
```

Receiver Variable

Offset	Field	Value
0	Bytes returned Bytes available Entry count	48 48 3
12	Result type Type record count Reserved	TypeR 3
24	Result type Type Length	TypeDescR kZonedTE_E 1
36	Result type Total digits Fractional digits	DecimalR 1 0

WATCH Statement Example

C Program Fragment

Consider the C program fragment in Figure 7. Variable i defines an integer.

Figure 7. Program for Scalar Evaluate Example

```
Line  C Source

 1    int i = 29;
```

Input Buffer

```
WATCH i
```

Receiver Variable

Offset	Field	Value
0	Bytes returned Bytes available Entry count	83 83 4
12	Result type Watch results count Reserved	WatchR 4
24	Result type Watch number Watch length	WatchNumberR 1 4
36	Result type Expression text offset Expression text length	ExpressionTextR 60 1
48	Result type Expression value offset Expression value length	ExpressionValueR 62 20
60	String space	See note.
Note: iSPP:08006F0054001004		

Error Messages

Message ID	Error Message Text
CPF1938 E	Command is not allowed while serviced job is not active.
CPF1939 E	Time-out occurred waiting for a reply from the serviced job.
CPF1941 E	Serviced job has completed. Debug commands are not allowed.
CPF3C19 E	Error occurred with receiver variable specified.
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF7102 E	Unable to add breakpoint or trace.
CPF7E01 E	Pointer to receiver variable is NULL.
CPF7E02 E	Receiver variable length not valid.
CPF7E03 E	Pointer to input buffer is NULL.
CPF7E04 E	Input buffer length not valid.
CPF7E05 E	Input buffer is not as long as specified.
CPF7E06 E	Pointer to error code structure is NULL.
CPF7E07 E	Not enough space was provided for error code.
CPF7E08 E	Value of BytesProvided field is not correct.

CPF7E09 E	Value of BytesProvided field, &1, is not correct.
CPF7E10 E	Internal error occurred.
CPF7E11 E	Type error occurred.
CPF7E12 E	Identifier does not exist.
CPF7E14 E	Field does not exist.
CPF7E15 E	Syntax error occurred.
CPF7E16 E	Token error occurred.
CPF7E17 E	Structure type error occurred.
CPF7E18 E	Pointer type error occurred.
CPF7E19 E	Integral type error occurred.
CPF7E1A E	Enumerated type error occurred.
CPF7E1B E	Arithmetic type error occurred.
CPF7E1C E	Scalar type error occurred.
CPF7E1D E	Address type error occurred.
CPF7E1E E	Adding type error occurred.
CPF7E1F E	Subtracting type error occurred.
CPF7E20 E	Relational type error occurred.
CPF7E21 E	Equality type error occurred.
CPF7E22 E	Casting type error occurred.
CPF7E23 E	Assignment type error occurred.
CPF7E24 E	Line number not found.
CPF7E25 E	Array type error occurred.
CPF7E26 E	Subscript type error occurred.
CPF7E27 E	Format type error occurred.
CPF7E28 E	Type error occurred.
CPF7E29 E	Unsupported bit field alignment.
CPF7E2A E	String constants are not supported.
CPF7E2B E	Type compatibility error occurred.
CPF7E2C E	Too few array dimensions specified.
CPF7E2D E	Too many array dimensions specified.
CPF7E2E E	Incorrectly formed range expression.

CPF7E2F E	Range expression expands to exceed input buffer.
CPF7E50 E	Decimal type error occurred.
CPF7E51 E	Decimal size error occurred.
CPF7E52 E	Unsupported syntax.
CPF7E53 E	Assignment size error occurred.
CPF7E54 E	Integer type error occurred.
CPF7E55 E	Constant type error occurred.
CPF7E56 E	Identifier is ambiguous.
CPF7E57 E	Integer constant not valid.
CPF7E58 E	Compiler not valid.
CPF7E59 E	String type error occurred.
CPF7E5A E	Substring extends beyond end of string.
CPF7E5B E	Format length error occurred.
CPF7E5C E	Hexadecimal constant not valid.
CPF7E5D E	Decimal constant size error occurred.
CPF7E5E E	Integer constant size error occurred.
CPF7E5F E	Relational size error occurred.
CPF7E60 E	Constant not a decimal number.
CPF7E61 E	System cannot determine which expansion of the template to use.
CPF7E62 E	Watch cannot be set on this variable.
CPF7E63 E	Watch length is not valid.
CPF7E64 E	Clear watch number not found.
CPF8E03 E	Internal error occurred.
CPF8E04 E	Internal error occurred.
CPF8E05 E	Error on equal operator.
CPF8E06 E	Error on not-equal operator.
CPF8E07 E	Error on greater-than operator.
CPF8E08 E	Error on greater-than-or-equal-to operator.
CPF8E09 E	Error on less-than operator.
CPF8E0A E	Error on less-than-or-equal-to operator.
CPF8E0B E	Error on logical-and operator.

CPF8E0C E	Error on logical-or operator.
CPF8E0D E	Error on logical-exclusive-or operator.
CPF8E0E E	Error on logical-not operator.
CPF8E0F E	Error on add operator.
CPF8E10 E	Error on subtract operator.
CPF8E11 E	Error on negate operator.
CPF8E12 E	Error on multiply operator.
CPF8E13 E	Error on divide operator.
CPF8E14 E	Error on increment operator.
CPF8E15 E	Error on decrement operator.
CPF8E16 E	Error on modulo operator.
CPF8E17 E	Pointer not set for location referenced.
CPF8E18 E	Conversion error.
CPF8E19 E	Error on absolute value operator.
CPF8E1A E	Domain violation occurred.
CPF8E1B E	Domain violation occurred.
CPF8E1C E	Error on write operator.
CPF8E1D E	Error on shift operator.
CPF8E1E E	Error on operand value.
CPF8E1F E	Error on load constant operator.
CPF8E20 E	Error on load address operator.
CPF8E21 E	Error on store indirect operator.
CPF8E22 E	Error on move operator.
CPF8E23 E	Error on fill operator.
CPF8E24 E	Incorrect array index value.
CPF8E25 E	Call stack entry does not exist.
CPF8E26 E	Translation failed.
CPF8E27 E	Call to user method failed.
CPF8E28 E	Variable not available to display.
CPF8E29 E	Debug recursion error.
CPF8E2A E	Error occurred while processing operation.

CPF8E2B E	Watch cannot overlap another active watch.
CPF8E2C E	Maximum number of watches exceeded.
CPF9541 E	Not in debug mode.
CPF9542 E	View not found.
CPF9549 E	Error addressing API parameter.

Debug Language Statements

Debug language statements are the principal mechanism by which a programmer debugs a program. Programmers control the operation of a program by:

- Entering break statements to select where the program will stop
- Entering step statements to run one or more statements of the program under investigation
- Entering watch statements to stop the program when a specified storage location is changed

The clear statement enables programmers to remove a particular breakpoint or all breakpoints. It can also be used to clear watches. Information about the state of the program being debugged can be extracted when program processing is suspended. The evaluate statement permits programmers to display the value of an expression, or to display an aggregate, and to alter the value of a variable.

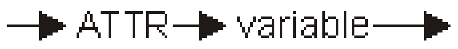
Debug language statements are constructed by the client program and placed in the input buffer. If multiple debug language statements are placed in the input buffer, they must be separated by one or more blanks. The Submit Debug Command API accepts the debug language statements of ATTR, BREAK, CLEAR, EVAL, QUAL, » SBREAK, « STEP, TBREAK, and WATCH.

When multiple debug statements are specified in the same input buffer, a QUAL statement must not follow an EVAL statement. The WATCH debug statement cannot be specified with any other debug statement, including another WATCH statement.

ATTR Statement

The variable appearing in an ATTR statement is found in the debug symbol table. The symbol table information for the variable is returned.

The following example shows an ATTR statement:

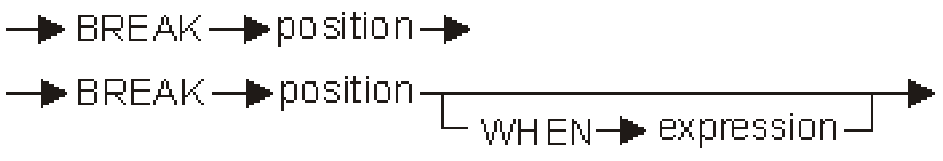


The locality of variables that appear in an ATTR statement is defined by the most recently run qualify statement. The program calling this API is advised to issue a qualify statement that defines the stop position when program operation is suspended.

Break Statement

The break statement permits a programmer to enter a breakpoint. Breakpoints are entered on the program under investigation. When the program under investigation encounters a breakpoint, operation is suspended.

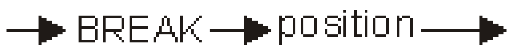
The following example shows a break statement:



The position marks where program operation is suspended when a breakpoint is encountered. Line numbers are used to identify the position when the break statement is entered. The line number entered is mapped to a statement by the Submit Debug Command API. A breakpoint causes the program to stop just before the break statement is run.

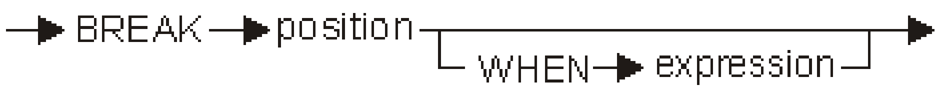
Unconditional and conditional breakpoints can be entered. Unconditional breakpoints are discussed first, followed by a discussion of conditional breakpoints.

An unconditional breakpoint is entered by issuing the first form of the break statement.



A line number is entered for the position. Line numbers are associated with each view in that they identify the lines of source in a view. Line numbers are assigned sequentially beginning with line one.

A conditional breakpoint is entered by issuing the second form of the break statement.



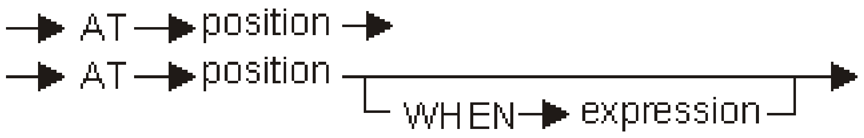
The position of a conditional breakpoint is assigned in the same way as the position in an unconditional breakpoint. A line number is entered for the position.

The condition of a conditional breakpoint is the expression following the reserved word WHEN. The result of the expression must have a Boolean or a logical value when evaluated. The expression is interpreted before the statement on which the breakpoint was entered is run. If the value of the expression is TRUE, operation of the program investigation is suspended. If the value of the expression is FALSE, operation continues without interruption.

The locality of variables used in the conditional expression is defined by the line number that defines the position.

A breakpoint can be replaced by entering another breakpoint using the same position. The most recent breakpoint entered on a position is the active breakpoint.

BREAK may be replaced by the reserved word AT in the statement that defines the break statement.

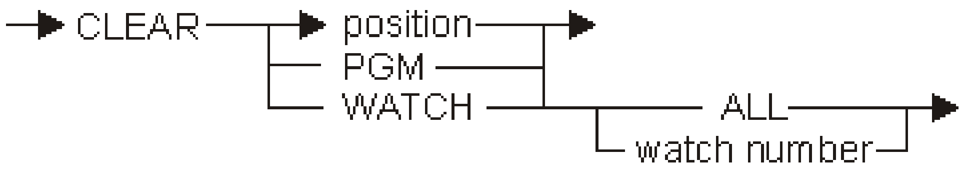


For threaded applications, breakpoints that are specified with the break statement are global to all threads in the job being debugged. These are called job breakpoints. Thread-specific breakpoints are set with the tbreak statement. A job may not have both a job breakpoint and thread breakpoints at the same position. When a job breakpoint is in effect and a thread breakpoint is specified, the job breakpoint is replaced. When thread breakpoints are in effect and a job breakpoint is specified, the thread breakpoints are replaced.

Clear Statement

The clear statement enables a programmer to remove a particular breakpoint or all breakpoints in a program. Particular breakpoints are identified by the number of the line on which they are active. All breakpoints in a program are designated by the keyword PGM. The clear statement is also used to clear one or all watch conditions. The keyword WATCH followed by the ALL keyword clears all watch conditions. If a watch number is specified after the WATCH keyword, only the watch represented by that watch number is cleared.

The following example shows a clear statement:

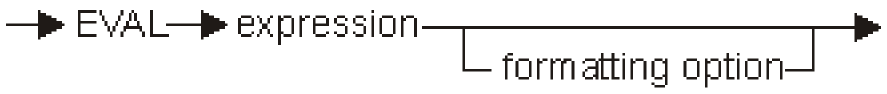


For threaded applications, if a thread breakpoint is in effect at the position specified, it is cleared in the current thread only. If the breakpoint is a job breakpoint, it is cleared for the job. When the clear statement with the PGM keyword is specified, it will remove all job and thread breakpoints.

Evaluate Statement

The expression appearing in an evaluate statement is evaluated, and the value of the expression is returned. The value of an expression is formatted according to the expression type.

The following example shows an evaluate statement:



An evaluate statement allows a programmer to display the value of an expression or an aggregate, or to alter the value of a variable. The precise definition of what can be displayed or altered is dependent on the language of the module being debugged.

Variables can be displayed or altered when program processing is suspended. Program operation is temporarily suspended as a result of encountering a breakpoint or completing a step statement. It is also suspended when a watch condition is satisfied.

Variables are formatted according to their type recorded in the HLL symbol table, and according to the language of the module being debugged. Formats available include integer, hexadecimal, exponential, and address, among others.

Variables also may be formatted using the formatting option. The formatting option has the general form of: *:<format code> <length>*, such as EVAL STRING :s 50.

The*:<format code>* can be one of the following:

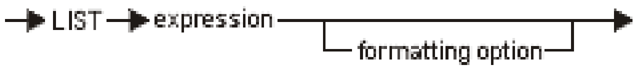
Format Code	**Description**
:c	An EBCDIC single-character format
:x	A hexadecimal format
:s	An EBCDIC character-string format (only for the C and C++ languages)
:f	An EBCDIC character-string format (only for the C and C++ languages). This returned type can be used by the source debugger to indicate that alternative formatting was requested by the user.
:a	An ASCII character-string format (only for the ILE languages). The string is converted from the job CCSID's related-ASCII CCSID to the job CCSID.
:u	A Unicode character-string format (only for ILE languages). The string is converted from Unicode CCSID 13488 to the job CCSID.

The *<length>* is a positive integer that indicates the number of bytes to format. The defaults for the format codes are as follows:

Format Code	**Default**
:c	1
:x	The length of the expression value
:s	30
:f	1024
:a	1024
:u	1024

The locality of variables that appear in an evaluate statement is defined by the most recently run qualify statement. The program calling this API is advised to issue a qualify statement that defines the stop position when program operation is suspended.

EVAL may be replaced by the reserved word LIST in the statement that defines the evaluate statement.



The following table describes the formatting of data by type.

Presentation Formats		
Type	**Format**	**Example**

kChar__8_E	c	A
kChar_16_E	Shift-out cc... shift-in	
kEnum____E	ccccc (dd)	yellow (25)
kString__E	ccccccccc	Hello World
kInt_32__E	-dd...d dd...d	-676
kPacked__E	dd.ddd -dd.ddd	5678.01
kZonedTE_E	dd.ddd -dd.ddd	5678.01
kZonedTS_E	dd.ddd -dd.ddd	5678.01
kZonedLE_E	dd.ddd -dd.ddd	5678.01
kZonedLS_E	dd.ddd -dd.ddd	5678.01
kBinD_16_E	dd.ddd -dd.ddd	5678.01
kBinD_32_E	dd.ddd -dd.ddd	5678.01
kBinD_64_E	dd.ddd -dd.ddd	5678.01
kFixedL__E	ccccc	Hello World
kHex_____E	xx xx xx xx	F1 F2 F3
kCard_32_E	dd...d	546
kReal_64_E	+d.d...dE+dd -d.d...dE-dd	-1.2345678901234E-95
KSpcPtr__E	Pointer types: BEP (behavior) IVP (invocation) LBL (label) MTP (method) OBP (object) PRP (procedure) SPP (space) SYP (system)	SPP:*NULL IVP:COFE001001201003 SPP:COCE100201021003

For threaded applications, the EVAL statement is run in the current thread.

Locality

Locality is the term used to describe the range over which an entity may be referred to in a module. The terms locality and scope are synonymous. By this definition, the locality of an entity is always confined to the compilation unit in which it was declared.

Entity is a formal way of describing all things that can be declared in a module. Variables, procedures, labels, types, and constants are entities.

The locality of an entity is defined by the block in which it is declared. An entity is visible in the block in which it is declared and all subordinate blocks. A variable can be referred to in the block in which it is declared.

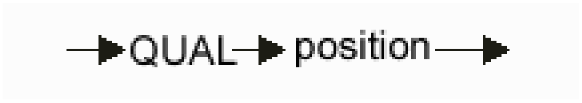
An entity may be declared in a block that encloses other blocks. The entity declared in the outer, enclosing block is visible in inner blocks if the name does not collide with other entities in inner blocks. A variable declared in an outer block can be referred to in an inner block if no variable of the same name is declared in the inner block.

To fully qualify a particular locality in a program, both program and module must be identified.

Qualify Statement

The qualify statement permits a programmer to define the locality of variables that appear in succeeding evaluate statements. Locality is defined by the line number operand on the qualify statement. The locality assigned is that block in which the line number appears.

The following example shows a qualify statement:



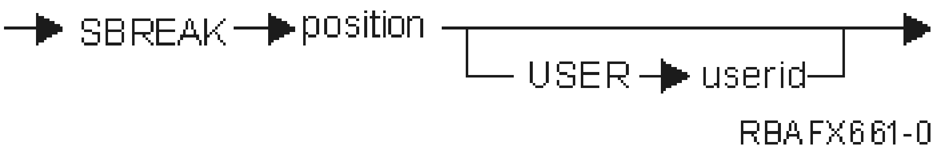
The locality assigned when a qualify statement is issued remains in effect until the next qualify statement is issued. The Submit Debug Command API keeps the locality assigned for the purpose of evaluating expressions. Users of the Submit Debug Command API are advised to issue the qualify statement whenever program operation is suspended. Use the line number of the stopped position to identify the current locality. In this way, programmers may issue several evaluate statements that refer to variables that are defined in the locality of the stopped position.

For threaded applications, the QUAL statement is run in the current thread.

SBreak Statement

The sbreak statement permits a programmer to enter a service entry breakpoint. Service entry breakpoints are entered on the program about to be spawned by another program. When the spawned program encounters a service entry breakpoint, operation is suspended.

The following example shows a sbreak statement:



The position marks where program operation is suspended when a service entry breakpoint is encountered. Line numbers are used to identify the position when the sbreak statement is entered. The line number entered is mapped to a statement by the Submit Debug Command API. A service entry breakpoint causes the program to stop just before the sbreak statement is run.

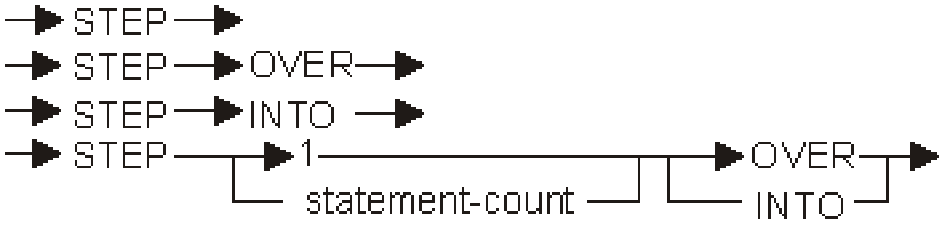
The userid specifies the user profile under which a job must be executing for the service entry point being set to be active in that job. If the userid is not specified, it defaults to the user profile under which the job in which the sbreak command is issued is running.

A service entry point, job breakpoint or thread breakpoint cannot exist at the same time at the same position. Only one of the three types of breakpoints may exist at a specified position. If the sbreak command is issued for a position in which one of the three types of breakpoints already exists, the existing breakpoint will be replaced by the service entry point.«

Step Statement

The step statement permits a programmer to run one or more statements of the program under investigation for testing purposes. The program being tested runs the number of statements specified in the statement-count operand. Operation of the program under test is suspended at completion of the step statement.

The following example shows a step statement:



If no value is entered for the statement-count, one statement is run.

The reserved words OVER and INTO direct the source debugger to step over or into procedures, respectively. If OVER appears in a step statement, the source debugger does not suspend operation in any procedures that are called. Procedures and functions are run without interruption.

The INTO reserved word directs the source debugger to stop in procedures that are called.

If neither INTO or OVER is entered on the step statement, OVER is assumed.

There are some step limitations. The following code cannot be entered using the step statement:

- Procedures in modules that have no debug data.
- Modules that are optimized at level 40.

For threaded applications, the STEP statement is run in the current thread. Each thread can step independently of each other, at the same time.

TBreak Statement

The tbreak statement permits a programmer to enter a breakpoint for the current thread. Breakpoints are entered on the program under investigation. When the program under investigation encounters a breakpoint in the thread, operation is suspended. The tbreak statement has the same format and operation as the break statement.

Each thread in a threaded application may have a different thread breakpoint at the same position. Job breakpoints and thread breakpoints cannot coexist.

A tbreak statement entered at the same position as a tbreak that was specified earlier in the same thread is replaced by the new thread breakpoint.

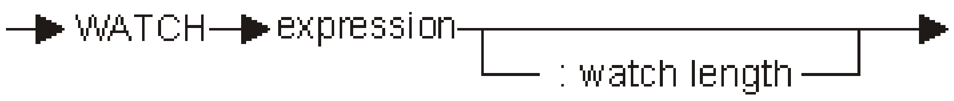
A tbreak statement entered at the same position as a job breakpoint that was specified earlier replaces the job breakpoint with a thread breakpoint.

A break statement entered at the same position as thread breakpoints that were specified earlier replaces all thread breakpoints at that position with a job breakpoint that is in effect for all threads.

Watch Statement

The watch statement permits a programmer to request a breakpoint when the content of a specified storage location is changed from its current value. After the watch condition is successfully set, a change to the content of the watched storage location causes program operation to be suspended. Then the Program Stop Handler exit program that is specified on the Start Source Debug API is called.

The following shows the syntax of the watch statement:



The expression is used to determine the address of the storage location to watch. The expression must resolve to an lvalue (that is, a location that can be assigned to). If an expression is specified that is not supported, error code CPF7E62 is returned. The scope of the expression variables in a watch statement is defined by the most recently issued QUAL debug language statement.

The length of the watch comparison operation is the same as the expression type length, or the length specified with the optional watch length parameter. For example, if a 4-byte binary integer is specified without the watch length parameter, the comparison length is 4 bytes. If the watch length parameter is specified, it overrides the length of the expression in determining the watch length. The watch length specification format is a colon character followed by the length in bytes to watch. For example, the watch command below would watch 2 bytes starting at the first byte of variable i:

watch i : 2

The watch length must be in the range 1 through 128 bytes. If the watch length is not valid, error code CPF7E63 is returned.

The maximum number of watches that can be active across the entire system is guaranteed to be at least 128, but may range up through 256, depending on how the watched storage is mapped by the system. This includes dedicated service tools (DST) watches. Exceeding this number results in error code CPF8E2C being returned. A user session may have as many watches as are available.

There are some restrictions on overlapping watch locations. If any of the following conditions are true, error code CPF8E2B is returned:

- Watches in same job: Two watch locations may not overlap in any way.
- Watches in different jobs: Two watch locations may not start at the same storage address. Otherwise, overlap is permitted.

A watch condition is cleared by using the CLEAR debug language statement.

It is important to understand that the watch statement establishes the watched storage location address when the watch statement is entered, and it does not change. This can cause misleading results if a temporary storage location is watched and that storage location is reused while the application is running. An example of this is the automatic storage of an ILE C procedure, which can be reused if the procedure ends.

The WATCH debug statement cannot be specified with any other debug statement, including another WATCH statement.

For threaded applications, the WATCH statement is run in the current thread. The address watched is global to all threads. Any thread changing a watched location will cause a breakpoint in that thread.

API introduced: V2R3

Top | Debugger APIs | APIs by category

Dump Module Variables (QteDumpModuleVariables) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Receiver variable length	Input	Binary(4)
3	Format name	Input	Char(8)
4	Qualified program name	Input	Char(20)
5	Program type	Input	Char(10)
6	Module name	Input	Char(10)
7	Data option	Input	Binary(4)
8	Continuation handle	Input	Char(16)
9	Error code	I/O	Char(*)

Service Program: QTEDMPV

Default Public Authority: *USE

Threadsafe: No

The Dump Module Variables (QteDumpModuleVariables) API is used to get a list of all the variable names and current values of those variables. Variable values may only be requested if an active call stack entry for the module specified exists 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 has a current call stack entry. All variables that were defined by the compiler and stored in the module HLL symbol table will be returned. This API supports the ILE CL, ILE COBOL, and ILE RPG compilers.

The module for which variable information is being requested must contain debug data. See the debug view (DBGVIEW) parameter of the Create RPG Module (CRTRPGMOD), Create COBOL Module (CRTCBLMOD), or Create CL Module (CRTCLMOD) command. It is not necessary that the job in which the program is running be in debug mode to use this API.

Variable names and, optionally, their values will be provided within the block in which they were declared. This API does not guarantee that those variables are returned in any particular order within the block.

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The variable that is to receive the list of program variables and current values for the specified module.

Receiver variable length

INPUT; BINARY(4)

The length of the receiver variable that is provided in the previous parameter. This value must be at least 48 to provide space for the receiver variable header section. The bytes available field tells the caller what size is required to receive the entire results of the request.

Format name

INPUT; CHAR(8)

The format of the information returned for the module. The possible format name is:

DMPV0100	Dump module variables.

Qualified program name

INPUT; CHAR(20)

The name of the program for which the variables and values will be provided.

The first 10 characters contain the name of the program. The second 10 characters contain the name of the library where the program is located. Each name will be left-justified. The special values of *LIBL and *CURLIB may be specified.

Program type

INPUT; CHAR(10)

The object type of the program. The possible values are:

**PGM*	ILE program
**SRVPGM*	ILE service program

This API cannot be used to dump variable information for an OPM program.

Module name

INPUT; CHAR(10)

The name of the module (left-justified) within the program. The module must be written in one of the supported ILE languages or an error is reported.

Data option

INPUT; BINARY(4)

The content of the information returned for the module. The possible values are:

0	Variable names only.
1	Variable names and current values in default character format (the type associated with the variable will be used in determining the format of the value returned).
2	Variable names, the current values in default character format, and the current values in hex format.

Continuation handle

INPUT; CHAR(16)

The handle used to continue from a previous call to this API that resulted in partially complete information. You can determine if a previous call resulted in partially complete information by checking the continuation handle variable in the receiver variable header section following the API call.

If the API is not attempting to continue from a previous call, this parameter must be set to blanks. Otherwise, a valid continuation value must be supplied. When continuing, the first entry in the returned receiver variable parameter is the entry that immediately follows the last entry returned in the previous call.

An error will occur under the following conditions:

- The continuation handle is not blank on the first request for a given set of input parameters.
- The continuation handle is not the same as provided in the receiver variable header section on the previous call to this API.

Error code

I/O; CHAR(*)

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

Format of the Receiver Variable

The receiver variable area consists of:

- A receiver variable header section.
- A module variable header section for each variable returned by the Dump Module Variables API. The module variable header section consists of:
  - A fixed-length header section
  - A variable-length section containing the information requested for the module variable.

Receiver Variable Header Section

Table 1. Receiver Variable Header Section

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of variable sections
12	C	CHAR(10)	Returned library
22	16	CHAR(10)	Reserved
32	20	CHAR(16)	Continuation handle
Note: The following information is repeated as many times as the value specified in the number of variable sections field.			
			Module variable header section

			Module variable section

Module Variable Header Section

This table describes the common header area to each subsequently defined module variable section.

Table 2. Module Variable Header Section

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Length of module variable section
4	4	BINARY(4)	Offset to next variable
8	8	BINARY(4)	Variable entry type

This portion of the module variable section will always start in the next available 4-word boundary to ensure proper alignment of the BINARY(4) fields within each section. The caller must use the offset to next variable field to find the start of the next module variable section and use the length of module variable section to determine the length of the current section.

Module Variable Section (Scalar Variable Entry Type)

The following table is used when the variable entry being returned is scalar. This section could occur by itself or following an array definition.

Table 3. Scalar Variable Section

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Variable type
4	4	BINARY(4)	Total digits
8	8	BINARY(4)	Precision
12	C	BINARY(4)	Scaling factor
16	10	BINARY(4)	Offset to variable name
20	14	BINARY(4)	Length of variable name
24	18	BINARY(4)	Length of default value
28	1C	BINARY(4)	Length of hexadecimal value
32	20	BINARY(4)	String content descriptor
36	24	BINARY(4)	Length of string prefix
		CHAR(*)	Variable name
		CHAR(*)	Default value
		CHAR(*)	Hexadecimal value

All variable values will be returned in displayable character format. For example, if the internal representation of a 2-byte unsigned integer is X'0345' the data returned through this API in the default value area will be '837 ' (X'F8F3F7404040'), and in the hex value area will be '0345' (X'F0F3F4F5').

When the scalar values of an array are being retrieved, the values will be returned in row major order, with no separating characters. The data option parameter will be used to determine if any values are displayed and in what form.

0	No values will be returned.
1	Only the default value of each scalar will be returned. The length of default value field will specify the length of each value. Each scalar value in the array will be provided in row major order.
2	The default value and the hex value of each scalar will be returned. The length of default value field and the length of hex value field will specify the length of each value. Each scalar value in the array will be provided with each representation in row major order with the default value leading each pair of values.

Module Variable Section (Array Definition Entry Type)

The following table is used when the variable entry being returned is an array. This section will define the array and will be followed by one or more scalar variable sections.

Table 4. Array Definition Variable Section

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Number of scalar fields
4	4	BINARY(4)	Offset to first variable
8	8	BINARY(4)	Offset to dimensions
12	C	BINARY(4)	Offset to array name
16	10	BINARY(4)	Number of array dimensions
20	14	BINARY(4)	Length of array name
		BINARY(4)	Dimension lower bound
		BINARY(4)	Dimension upper bound
		CHAR(*)	Array name

Module Variable Section (Block Definition Entry Type)

The following table is used when the variable entry being returned is a block definition. One of these sections will exist for each block defined in the program. A block definition entry will precede all other variable entry sections for variables defined within the specified block.

Table 5. Block Definition Variable Section

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Block number
4	4	BINARY(4)	Offset to block name
8	8	BINARY(4)	Length of block name

		CHAR(*)	Block name

Field Descriptions

Array name. The field containing the name of the array.

Block name. The field containing the name of the block.

Block number. The number of the block.

Bytes available. The number of bytes of data available to be returned. All available data is returned if enough space is provided.

Bytes returned. The number of bytes of data returned.

Continuation handle. When not all the requested data can be returned on a single call to this API, a value will be supplied in this field which may be used to continue on the next call to this API.

Default value. The value of the variable represented in the default format for the variable type.

Dimension lower bound. The lower bound of an array dimension.

Dimension upper bound. The upper bound of an array dimension.

Hexadecimal value. The value of the variable represented in hexadecimal format as it is stored in the machine.

Length of array name. The length of the array name field.

Length of block name. The length of the block name field (may be zero if no name is associated with the block).

Length of default value. The length of the data in the default value field. This will be zero if the data option parameter is 0.

Length of hexadecimal value. The length of the data in the hexadecimal value field. This will be zero if the data option parameter is 0 or 1.

Length of module variable section. The module variable entry section length, including the length of the module variable section header.

Length of string prefix. The length of the string prefix (may be 0 if no prefix is associated with the string).

Length of variable name. The length of the variable name field.

Number of array dimensions. The number of dimensions in the array. The dimension upper and lower bound fields are repeated for each array dimension.

Number of scalar fields. Number of scalar fields in each array element. There will be one module variable section for each scalar following an array definition header.

Number of variable sections. The number of variable entries returned by the API. These include block variable entries, scalar variable entries, and array variable entries.

Offset to array name. Offset to the start of the array name field.

Offset to block name. Offset to the start of the block name field.

Offset to dimensions. Offset to the start of the first dimension lower bound field.

Offset to first variable. Offset to the start of the module variable header section for the first scalar variable.

Offset to next module variable header section. Offset to the start of the next module variable header section.

Offset to variable name. Offset to the start of the variable name field.

Precision. The precision associated with a decimal type (packed, zoned, or binary decimal).

Reserved. An ignored field.

Returned library. The library where the program was found. This is useful when *LIBL or *CURLIB is specified for the program library portion of the program name parameter.

Scaling factor. The scaling factor associated with a decimal type (packed, zoned, or binary decimal).

String content descriptor. The type of the string variable. It may be one of the following values:

0	An error occurred evaluating the variable
1	A null-terminated unicode string
2	A length-prefix-2 unicode string
3	A length-prefix-4 unicode string
4	A fixed-length unicode string
5	A variable-length unicode string
6	A null-terminated graphic string
7	A length-prefix-2 graphic string
8	A length-prefix-4 graphic string
9	A fixed-length graphic string
10	A variable-length graphic string
11	A date string
12	A packed date string
13	A time string
14	A packed time string
15	A timestamp string

Total digits. The total number of digits associated with a decimal type (packed, zoned, or binary decimal).

Variable entry type. The type of variable section that follows the module variable header section. It may be one of the following values:

0	Scalar variable
1	Array definition
2	Block definition

Variable name. The field containing the name of the variable.

Variable type. The data type of the variable. It may be one of the following values:

0	An error occurred evaluating the variable
1	An 8-bit (1-byte) character
2	A 16-bit character
3	A 32-bit quantity having ordinal values of zero or one. Zero is the ordinal value for FALSE, and one is the ordinal value for TRUE.
4	A 16-bit unsigned integer
5	A 32-bit unsigned integer
6	A 16-bit two's complement (signed) integer
7	A 32-bit two's complement (signed) integer
8	A 32-bit IEEE 754 floating point value
9	A 64-bit IEEE 754 floating point value
10	A 128-bit space pointer
11	A fixed-length character string
12	A packed decimal
13	A zoned trailing embedded sign
14	A zoned leading embedded sign
15	A zoned trailing separate sign
16	A zoned leading separate sign
17	A 16-bit binary decimal
18	A 32-bit binary decimal
19	A 64-bit binary decimal
20	A 32-bit index value
21	An 8-bit unsigned integer
22	An 8-bit signed integer
23	A 64-bit unsigned integer
24	A 64-bit signed integer
25	A variable-length character string

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.
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: V3R1

Top | Debugger APIs | APIs by category

Retrieve Program Variable (QTERTVPV) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	Program variable	Input	Char(132)
4	Basing pointer	Input	Array(5) of Char(132)
5	Starting position	Input	Binary(4)
6	Length of string	Input	Binary(4)
7	Output format	Input	Char(10)
8	Program	Input	Char(10)
9	Recursion level	Input	Binary(4)
10	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Retrieve Program Variable (QTERTVPV) API retrieves the current value of one program variable in a program that is being debugged. The information is returned to the calling program in a receiver variable. The amount of returned information is limited to the size of the receiver variable. This information is similar to the information returned using the Display Program Variable (DSPPGMVAR) command.

Restriction

This API is valid only in debug mode and supports original program model (OPM) programs only. It cannot be used if the user is servicing another job and that job is on a job queue, or is held, suspended, or ended.

Required Parameter Group

Receiver variable

OUTPUT; CHAR(*)

The variable that is to receive the information requested. The minimum size for this area is 8 bytes. If the size of this area is smaller than the available information, the API returns only the data that the area can hold.

See Format of Receiver Variable for details about the format.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. If this value is larger than the actual size of storage allocated for the receiver variable, the results are not predictable. The minimum length is 8 bytes.

Program variable

INPUT; CHAR(132)

The name of the program variable whose value is to be retrieved. Possible values follow:

**CHAR*	This special value is specified instead of a variable name if a basing pointer is also specified. This special value returns a character view of the area addressed by a pointer.
Program variable name	The name of the program variable. For information about program variables, see the topic on program-variable description in the Control Language (CL) information.

Basing pointer

INPUT; ARRAY(5) of CHAR(132)

In languages where a program variable may be based on a pointer variable, you can specify the basing pointers for the variable to be retrieved. Up to five basing pointers may be specified. If the basing pointer is an element in an array, the subscript representing an element in the array must be specified. Up to 132 characters can be specified for one basing pointer name. If no basing pointer is specified, then the structure must be initialized to blanks. If one or more basing pointers are specified, then the subsequent array entries must be initialized to blanks. For more information on basing pointers, refer to the topic on basing-pointer description in the Control Language (CL) information in the iSeries Information center.

Starting position

INPUT; BINARY(4)

For string variables only, the starting position in the string from which its value is being retrieved. For a bit string, the value is the starting bit position. For a character string, the value is the starting character position.

This parameter is ignored on nonstring variables but must be initialized to any number greater than 0.

Length of string

INPUT; BINARY(4)

For string variables only, the length of the string retrieved, starting at the position specified by the start parameter. For a bit string, this value is the number of bits to retrieve. For a character string, this value is the number of characters to retrieve.

0	The value of the string variable is retrieved to the end of the string or retrieved for 200 bytes, whichever is less. If the string variable has a maximum length of zero, only 0 is allowed.
Retrieve length	The length of data to retrieve. This parameter is ignored on nonstring variables but must be initialized to any number 0 or greater.

Output format

INPUT; CHAR(10)

The format in which the value is to be returned.

**CHAR*	The value of the program variable is returned in character form.
**HEX*	The value of the program variable is returned in hexadecimal form.

Program

INPUT; CHAR(10)

The name of the program that contains the program variable to be retrieved.

**DFTPGM*	The program currently specified as the default program will be used.
Program name	The name of the program whose program variable is retrieved.

Recursion level

INPUT; BINARY(4)

The recursion level of the program that contains the program variable.

0	The last (most recent) call of the program is the one from which the automatic program variable is retrieved.
n	The number of the recursion level of the program from which the automatic program variable is retrieved.

This parameter is ignored on static variables but must be initialized to any number 0 or greater.

Error code

I/O; CHAR(*)

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

Format of Receiver Variable

The following table shows the information supplied in the receiver variable parameter. For more information on each field, see Field Descriptions.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Variable type
12	C	BINARY(4)	Data error
16	10	POINTER	Pointer to variable
32	20	BINARY(4)	Bit position
36	24	BINARY(4)	Variable length
40	28	BINARY(4)	Variable precision
44	2C	BINARY(4)	Number of array dimensions

48	30	BINARY(4)	Number of array elements returned
52	34	ARRAY(15) of BINARY(4)	Subscript bounds
172	AC	BINARY(4)	Element length
176	B0	BINARY(4)	Character string length
180	B4	CHAR(64)	Reserved
244	F4	CHAR(*)	Data retrieved

The following tables show the information supplied in the data retrieved field. The variable type field, which is enclosed in parentheses, indicates which table is used.

Data for Binary Numeric (1)

Offset			
Dec	**Hex**	**Type**	**Field**
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(*)	Variable value

Data for Floating Point (2)

Offset			
Dec	**Hex**	**Type**	**Field**
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(*)	Variable value

Data for Zoned Decimal (3)

Offset			
Dec	**Hex**	**Type**	**Field**
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(*)	Variable value

Data for Packed Decimal (4)

Offset			
Dec	Hex	Type	Field
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(*)	Variable value

Data for Fixed Character (5)

Offset			
Dec	Hex	Type	Field
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(*)	Variable value

Data for Varying Character (6)

Offset			
Dec	Hex	Type	Field
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	BINARY(4)	Varying character length
256	100	CHAR(*)	Variable value

Data for Fixed Bit (7)

Offset			
Dec	Hex	Type	Field
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(*)	Variable value

Data for Unsigned Binary (8)

Offset			
Dec	**Hex**	**Type**	**Field**
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(*)	Variable value

Data for Space Pointer (9)

Offset			
Dec	**Hex**	**Type**	**Field**
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(8)	Hexadecimal offset
260	104	CHAR(8)	Reserved
268	10C	CHAR(30)	Object addressed by pointer
298	12A	CHAR(10)	Library name
308	134	CHAR(8)	Object type

Data for Data Pointer (10)

Offset			
Dec	**Hex**	**Type**	**Field**
244	F4	CHAR(7)	Message ID
251	FA	CHAR(1)	Reserved
252	FB	CHAR(8)	Hexadecimal offset
260	104	CHAR(30)	Object addressed by pointer
290	122	CHAR(10)	Library name
300	12C	CHAR(8)	Object type
308	134	BINARY(4)	Data type
312	138	BINARY(4)	Data length
316	13C	BINARY(4)	Data precision
320	140	BINARY(4)	Data string length
324	144	BINARY(4)	Element length
328	148	CHAR(*)	Data

Data for Instruction Definition List (11)

Offset			
Dec	Hex	Type	Field
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(8)	Instruction number
260	104	CHAR(8)	Reserved
268	10C	CHAR(30)	Object addressed by pointer
298	12A	CHAR(10)	Library name
308	134	CHAR(8)	Object type

Data for System Pointer (12)

Offset			
Dec	Hex	Type	Field
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(16)	Authorization
268	10C	CHAR(30)	Object addressed by pointer
298	12A	CHAR(10)	Library name
308	134	CHAR(8)	Object type

Data for Machine Space Pointer (13)

Offset			
Dec	Hex	Type	Field
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(8)	Hexadecimal offset
260	104	CHAR(8)	Reserved
268	10C	CHAR(30)	Object addressed by pointer
298	12A	CHAR(10)	Library name
308	134	CHAR(8)	Object type

Data for Exception Description (14)

Offset			
Dec	**Hex**	**Type**	**Field**
244	F4	CHAR(7)	Message ID
251	FB	CHAR(1)	Reserved
252	FC	CHAR(1)	Control
253	FD	CHAR(1)	Handler type
254	FE	CHAR(8)	Instruction number
262	106	CHAR(10)	Program name
272	110	CHAR(10)	Library name
282	11A	CHAR(2)	Reserved
284	11C	BINARY(4)	Compare string length
288	120	CHAR(28)	Compare string
316	13C	CHAR(1)	Job log
317	13D	CHAR(3)	Message type
320	140	BINARY(4)	Number of message IDs
324	144	ARRAY(*) of CHAR(7)	Array of messages

Field Descriptions

Array of messages. An array of the number of message IDs is returned.

Authorization. Pointer authorization.

Bit position. The starting bit position, 1-8, for bit strings returned in *HEX format. The least significant bit is 1 and 8 the most significant bit. This field will be initialized to 0 for any other variable type.

Bytes available. The number of bytes of data available to be returned. All available data is returned if enough space is provided.

Bytes returned. The number of bytes of data returned.

Character string length. For output format *CHAR, this value is the length of the returned character string. For output format *HEX, this value is initialized to 0. For fixed character, varying character, and fixed bit variables this field contains the actual length of the data returned for *CHAR and *HEX output formats. For pointers and exception monitors this field is 0.

Comparison string. The specified comparison string.

Comparison string length. The length of the comparison string. This value is 0 if a value is not specified.

Control. Exception monitor control action. The following values may be returned:

X'00'	Default
X'01'	Off

X'02'	Resignal
X'04'	Defer
X'05'	Handle

Data. The data addressed by the pointer. This field is returned in the corresponding output format for the variable type (data type).

Data error. Whether an error was returned when returning a variable.

0	No errors were returned with the variable data.
1	One or more errors were returned with the variable data.

Data length. The length of the data addressed by the pointer. This is the same value as in the variable length field in the header.

Data precision. The precision of the data addressed by the pointer. This is the same value as in the variable precision field in the header.

Data retrieved. If an error is encountered while retrieving the data, CPD messages may be returned instead of the variable data. The structure of this parameter is dependent on the object type. The format of the data depends on the variable type field.

Data string length. The string length of the data addressed by the pointer. This is the same value as in the variable string length field in the header.

Data type. The type of data addressed by the pointer. This is the same value as in the variable type field in the header.

Element length. The length of the data element returned. If this field is 0, each element can be a different length and the user must go to the element to get the element length.

Handler type. Exception monitor handler type.

'00'X	External handler
'01'X	Call internal handler
'02'X	Branch point handler

Hexadecimal offset. Hexadecimal offset of the space pointed to by the space or machine space pointer.

Instruction number. The exception handler instruction number for a monitor with an internal handler or X'0' for an external handler.

Job log. Put messages on job log.

0	No
1	Yes

Library name. The library containing the object addressed by the pointer, *LIBL, or X'0' for internal monitors.

Message ID. If an error was received with the variable data, this field contains the diagnostic message ID. If no error was received with the variable's data, this field contains blanks.

Message type. Message types being monitored.

100	Escape
010	Notify
001	Status

More than one message type can be monitored at a time. If the first and third characters are 1's, then escape and status messages are being monitored.

Number of array dimensions. If the variable is an array or an element of an array, this field is the number of array dimensions. Otherwise, this field is initialized to 0.

Number of array elements returned. If the variable is an array, this field is the number of array elements returned. Otherwise, this field is initialized to 0.

Number of message IDs. The number of message identifiers being monitored.

Object addressed by pointer. The fully qualified name of the object addressed by the pointer.

Object type. The Machine Interface (MI) type of the object addressed by the pointer.

Pointer to variable. Pointer to variable, if applicable. For example, a pointer is not returned to a variable of type machine space pointer or for an exception description. For system security reasons a pointer is not returned if the security level is 50 and if the job using the API is servicing and debugging another job.

Program name. External handler program name or X'0' for an internal monitor.

Reserved. An ignored field.

Subscript bounds. The subscript lower bounds and subscript upper bounds for each array dimension. If the variable is not an element of an array, or the dimension is not used, the subscript lower and upper bounds are initialized to 0.

Varying character length. The actual length of the varying character string.

Variable length. The length of the variable value. For bit strings, this value is the number of bits. For packed and zoned variables, this value is the number of digits. For pointers and exception monitors this field is 0. For all other variable types, this value is the number of bytes.

Variable precision. The number of decimal digits or fractional digits for zoned and packed variables. For any other variable type, this field will be initialized to 0.

Variable type. The following are the possible variable types:

1	Binary numeric
2	Floating point
3	Zoned decimal
4	Packed decimal
5	Fixed character
6	Varying character
7	Fixed bit
8	Unsigned binary
9	Space pointer
10	Data pointer

11	Instruction definition list
12	System pointer
13	Machine space pointer
14	Exception description

Variable value. The value of the variable being retrieved.

The following messages may be returned in this field:

CPD1901	Variable contains invalid decimal data.
CPD1902	Pointer to be displayed not set to any address.
CPD1903	Floating-point value displayed is not exact.
CPD1904	Object not found for system pointer with initial value.
CPD1905	Variable not found for data pointer with initial value.
CPD1906	Variable to be displayed contained in deleted object.
CPD1907	Variable refers to object with freed storage.
CPD1908	Space addressing error for variable.
CPD1909	Pointer alignment error. Pointer not on 16-byte boundary.
CPD1910	High-level language (HLL) pointer invalid.
CPD1911	START plus LEN values exceed length of string.
CPD1913	Space addressing error for variable.
CPD1914	Pointer addresses a deleted object.

Error Messages

Message ID	Error Message Text
CPF1902 E	No default program exists.
CPF1903 E	Program &1 not in debug mode.
CPF1905 E	Starting position parameter is not valid.
CPF1906 E	Command is not valid. No programs in debug mode.
CPF1915 E	Length parameter is not valid.
CPF1919 E	Recursion level parameter is not valid.
CPF1927 E	Output format name not valid.
CPF1938 E	Command is not allowed while serviced job is not active.
CPF1939 E	Time-out occurred waiting for a reply from the serviced job.
CPF1941 E	Serviced job has completed. Debug commands are not allowed.
CPF24B4 E	Severe error while addressing parameter list.
CPF3C19 E	Error occurred with receiver variable specified.

CPF3C24 E	Length of the receiver variable is not valid.
CPF7133 E	Variable or basing pointer name missing.
CPF9549 E	Error addressing API parameter.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

API Introduced: V2R3

Top | Debugger APIs | APIs by category