IBM i
7.4

Rational Open Access: RPG Edition

RZAS-M000-02

Note

Before using this information and the product it supports, read the information in "Notices" on page 63.

This edition applies to Rational® Open Access: RPG Edition 7.4 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Rational Open Access: RPG Edition

Open Access: RPG Edition allows RPG programmers to access new technologies within their RPG programs by specifying programs or service programs which connect the new technology with the RPG programs.

- Handlers for Open Access files. See "Rational Open Access: RPG Edition" on page 1.
- Parsers for the DATA-INTO operation code. See "Writing a parser for the RPG DATA-INTO operation code" on page 29.

What's New

New and changed features

The following is a list of enhancements made for each release of Open Access since 7.1:

- "What's New in this Release?" on page 1

You can use this section to link to and learn about new Rational Open Access: RPG Edition functions.

Note: The information for this product is up-to-date with the 7.2 release of the RPG compiler. If you are using a previous release of the compiler, or if you are using a previous target release, you will need to determine what functions are supported by that release of the compiler. For example, if you are compiling with target release V6R1M0, the functions new to the 7.2 release will not be supported.

What's New since 7.3?

This section describes the enhancements made to Open Access: RPG Edition since 7.3.

Information about writing a generator for the DATA-GEN operation code
See "Writing a generator for the RPG DATA-GEN operation code" on page 40.

Information about writing a parser for the DATA-INTO operation code
See "Writing a parser for the RPG DATA-INTO operation code" on page 29.

What's New in this Release?

This section describes the enhancements made to Open Access: RPG Edition in 7.2.

CCSID of alphanumeric data for Open Access files
Alphanumeric data can have a CCSID other than the job CCSID. Handlers that can support all CCSIDs for alphanumeric data must work with two new subfields of the handler parameter.

- *canHandleCcsids*
- *alphaCcsids*

Rational Open Access: RPG Edition

Rational Open Access: RPG Edition provides a way for RPG programmers to use the simple and well-understood RPG I/O model to access resources and devices that are not directly supported by RPG.

Open Access opens up RPG's file I/O capabilities, allowing anyone to write innovative I/O handlers to access other devices and resources such as:

- Browsers
- Mobile devices
- Cloud computing resources
- Web services

- External databases
- XML files
- Spreadsheets
- And more

An Open-Access application has three parts:

1. An RPG program that uses standard RPG coding to define an Open Access file and use I/O operations against the file.
2. A handler procedure or program that is called by Open Access to handle the I/O operations for the file.
3. The resource or device that the handler is using or communicating with.

Open Access is the linkage between parts 1 and 2.

Open-Access handlers

Open Access does not provide the handlers.

Anyone can write the handlers that extend RPG IV's I/O capabilities to new resources and devices.

- Software tool vendors
- Business partners
- Services organizations
- Users

The provider of the handler can choose the RPG device type whose I/O operations best fit the functions provided by the handler. For example, a user-interface application could map to a WORKSTN file, an Excel document could map to a PRINTER file, and a web service could map to a keyed DISK file.

Two ways to approach Open Access

1. The handler is written after the application is written.

   For example, an existing application that uses 5250 display files is modified to use Open Access for the WORKSTN files.

   - The RPG program is modified by adding the HANDLER keyword to the WORKSTN files
   - The handler must handle all the operations and requirements of the existing RPG program.
   - This type of handler will often be provided by an outside expert such as a software tool vendor or business partner.
2. The handler is written before the application is written.

   For example, the RPG programmer wants to use a web service that returns information for a specific set of criteria.

   - The handler provider creates a keyed database file matching the web service with a field for each piece of information returned by the web service, and a key field for each criterion needed by the web service. This file will not hold any data; it will only be used for defining externally-described files and data structures in the RPG program and the handler.
   - The handler provider can tell the RPG programmer what I/O operations that the handler will support. For example, it might only support OPEN, CHAIN, CLOSE.
   - The RPG programmer codes the RPG program using the file as an externally described keyed DISK file, with the HANDLER keyword to identify the Open-Access handler.
   - The handler uses externally-described data structures defined from the same file.
   - This type of handler might be written by the same RPG programmer who uses the Open-Access file, or it might be provided by an outside expert.

How Open Access works

When an RPG program performs an I/O operation for a system file, a system data management function is called to handle the operation.

When an RPG program performs an I/O operation for an Open Access file, the Open-Access handler is called. The handler receives a data structure parameter with subfields that enable the handler to perform the correct I/O operation, and provide information back to the RPG program.

If the file is externally-described, it must be available to the RPG compiler at compile-time. Open Access does not require the file to be present at run-time, but an individual handler may require the file to be present.

RPG coding to use Open Access

Other than the HANDLER keyword, there is no new RPG syntax related to using an Open-Access file.

HANDLER keyword

The RPG programmer indicates that a file is an Open-Access file by coding the HANDLER keyword on the File specification. All the operations allowed by RPG for the specified device (DISK, PRINTER, WORKSTN) are available for the Open-Access file.

The HANDLER keyword identifies the program or procedure that will handle all operations for the file.

```
Fmyfile    CF   E          WORKSTN HANDLER('MYLIB/MYSRV(hdlMyfile)')
```

Figure 1. Example of the HANDLER keyword

The HANDLER keyword has two parameters:

Handler

This keyword parameter can have one of the following values:

- A character literal or character variable identifying a procedure in a service program. The value must be in the form `'LIBRARY/SRVPGM(procedure)'` or `'SRVPGM(procedure)'`. The names are case-sensitive.
- A character literal or character variable identifying a program. The value must be in the form `'LIBRARY/PGM'` or `'PGM'`. The names are case-sensitive.
- A prototype for a bound procedure.
- A procedure pointer literal (%PADDR) or variable.

When the handler is identified by a prototype or by %PADDR, it is resolved at bind time of the RPG program. When the handler is identified by a character literal or character variable, it is resolved each time the Open-Access file is opened.

Parameter passed to the handler (optional)

This keyword parameter identifies an RPG variable to be passed to the handler from the RPG program.

This parameter can be used to pass additional information to the handler that is not available through an RPG file operation.

For example, to process a file in the Integrated File System, the handler needs to know the path to the file. The RPG programmer can provide this through the second HANDLER keyword parameter. The second parameter is the name of a variable, usually a data structure. In the example below, the variable is the data structure *ifsDs*. When the handler is called, it will receive a pointer to the *ifsDs* data structure.

The provider of the handler would tell the RPG programmer whether the additional information was required, and what data structure to use as a template. Usually, the provider of the handler would provide a template data structure in a /COPY file. In the example, the library MYLIB contains the handler service program HDL_IFS containing procedure *readIfs*, the database file HDL_IFS, and the /COPY File QRPGLESRC with member HDL_IFS. The copy file has the template data structure *hdlIfs_t*.

```
FmyIfsFile IF   E   DISK  EXTDESC('MYLIB/HDL_IFS')
F                         USROPN
F                         HANDLER(‘MYLIB/HDL_IFS(readIfs)
F                                : ifsDs)
 /copy MYLIB/QRPGLESRC,HDL_IFS
D ifsDs         DS        LIKEDS(hdlIfs_t)
 /FREE
    ifsDs.path = '/home/mydir/myIfsFile.txt';
    OPEN myIfsFile;
```

Figure 2. Example of the HANDLER keyword with two parameters

Comparison to RPG SPECIAL files

An Open-Access file is similar in nature to a SPECIAL file. A SPECIAL file also uses a user-written program to handle the operations for the file, and it allows additional parameters to be passed to the handler from the RPG program.

Some of the differences between Open-Access files and SPECIAL files are

- A SPECIAL file only allows the operations available for a sequential (SEQ) file. An Open-Access file can be defined for any type of RPG device, and can use all the operations available for that device.
- A SPECIAL file handler only receives a minimal amount of information about the file operation. An Open-Access file handler receives much more information such as the name of the file, record format, the names and types of the fields.
- A SPECIAL file handler can only pass back a minimal amount of feedback information: the result status of 0, 1 or 2, and a 5-digit SPECIAL error code value. If an error occurs for a SPECIAL file, the RPG status code is always 1231 (Error in SPECIAL file). An Open-Access handler has the ability to pass back much more information such as the RPG status code, the file feedback areas, the relative record number, function-key pressed, printer overflow.
- A SPECIAL file handler can only be a program. An Open-Access handler can be a program or a procedure.

Coding the Open Access handler

An Open Access handler can be coded in any ILE language.

Library QOAR has copy files defining the data structures and constants related to the handler parameter for ILE RPG, ILE C, and ILE COBOL. The member name in each source file (QRPGLESRC, H, QCBLLESRC) is QRNOPENACC.

Library QOAR is the library for product 5733-OAR. The source files can be used without having a license for the product.

Two modes for I/O data

There are two modes available for the handler to access and provide data. This applies to input and output data, and the search argument for keyed operations. It also applies to the indicator array for files that define the INDARA keyword.

The handler can choose which mode to use when it is processing the OPEN operation by setting the *useNamesValues* subfield of the handler parameter to '1'.

Name-value information

This mode is available only for externally-described files.

The input and output records and the search keys are provided as an array of information about each field. Any **IN* indicators associated with the record are included in the array of field information, whether or not the file is defined with the INDARA keyword. The information includes the external short name, the data type, the length, decimals, date and time format, the CCSID of the data, whether the field is null-capable, and whether the field has the null value. The data for each field is provided in a human-readable form. For example, numeric values are formatted as they would be by the %CHAR RPG built-in function; for example `'-1.23'`.

Most values are in the job CCSID, but UCS-2 and and DBCS values are in the CCSID used in the external file. Alphanumeric fields may also be in the CCSID used in the external file. See *alphaCcsids*.

When the value for a field is set by the handler, the handler must set the value in the same human-readable form that Open Access uses to provide the values for output-capable fields.

Remember:

1. The name-value information is available only for I/O to externally-described record formats and for input operations to externally-described files when there is only one record format in the file.
2. When name-value information has been requested by the handler, and the information is not available, either because the file is program-described, or because the operation is an input operation to a file with more than one record format, the request for name-value information will be ignored by RPG. No exception will be given; it is up to the handler to detect this situation. The handler may send an exception, or set the return code to a failing status, or the handler may adjust to the situation and use the record data information provided by the data structures matching the I/O buffers, key structures, and indicator array.
3. If the file uses INDARA, the relevant indicators appear as fields in the name-value array.

Data structures matching the I/O buffers

The data related to the I/O operation is passed to the handler in the form of I/O buffer data structures, null-indicator arrays, and INDARA indicator. These data structures and arrays are formatted exactly as they would be passed to data-management functions. No name and type information is passed to the handler. A handler that is specifically written to handle a particular file may use externally described data structures to access the buffers. A more generic handler that can handle many files may have to call an API to determine the information about the fields in the file.

If the file uses INDARA, the handler must work with the *indara* subfield of the handler parameter to use or set the values of any indicators used by the file.

If the RPG program uses null-indicators for the file, the null-indicator array for the record is in a separate buffer.

Open Access sets the pointers for all the buffers in the handler parameter. The handler is responsible for using or setting the data in the buffers.

Handling input data

Several operations require the handler to provide input data to the RPG program.

Using the *recordName* subfield

If the file is externally described, the RPG program must know which record format is associated with the input data.

- If the *recordName* subfield is not blank, the RPG program explicitly specified the required record format, or there is only one record format for the file. In this case, the handler must provide input data for the specified record format.
- If the *recordName* subfield is blank, the RPG program did not specify a specific record format. In this case, the handler must set the *recordName* subfield to indicate which record format is associated with the input data.

Using the *rrn* subfield

If relative record numbers are relevant for the file or record format, the handler may provide this value in the *rrn* subfield. The RPG programmer can obtain this value using the RECNO keyword on the File specification.

Handling input data using data structures

- Use the *inputBuffer* pointer subfield to locate the input buffer for the file and use the *inputBufferLen* subfield to determine the number of bytes provided for the input buffer.
- For an externally-described file, use the *inputNullMap* pointer subfield to access the input null map for the file and use the *inputNullMapLen* subfield to determine the number of bytes provided for the input null map. The input null map is an array of indicators with one element for each field in the record. For a field which is null-capable, a value of '1' indicates that the field has the null value, and a value of '0' indicates that the field does not have the null value. For a field which is not null-capable, the value should always be '0'. If the *inputNullMap* pointer is null, the input null map is not relevant for the file within the RPG program.
- For an externally-described file, set the value of each field in the input buffer according to the data type of the field, and if null-values are relevant for the file, set the indicator in the input null map for each null-capable field.
- For a program-described file, set the input buffer according to the definition of the record that is agreed upon by the RPG programmer and the handler provider.

Handling input data using name-value information

- Use the *namesValues* pointer subfield to access the array of information about each field in the record format.
- For each *input* field, or each *input* or *output* field in the case of a subfile record format
  - Use the *value* pointer subfield to locate the buffer available for the field.
  - Use the *valueMaxLenBytes* subfield to determine the maximum length available for the data.
  - Set the value of the field in the buffer, according to the data type of the field. See "Data types used in name-value information" on page 23 for more information.
  - Set the *valueLenBytes* subfield to indicate the number of bytes that were set by the handler in the buffer.
  - If the *isNullCapable* subfield indicates that the field is null-capable, set the *hasNullValue* to '1' if the field has the null value, and to '0' otherwise.

    Remember: The data for a field must be set to a valid value even if the *hasNullValue* subfield is set to '1'.

Handling output data

Several operations require the handler to use output data from the RPG program.

Using the *recordName* subfield

If the file is externally described, use the *recordName* subfield to determine which record format is associated with the output data.

Using the relative record number

The RPG programmer may use the RECNO to specify the relative record number for the new record. For system files, the relative record number may only be used to write to existing records which have been deleted. If the *rrn* subfield is greater than zero, then the RPG programmer intends the record to replace the deleted record at the specified relative record number.

Handling output data using data structures

- Use the *outputBuffer* pointer subfield to locate the output buffer for the file and use the *outputBufferLen* subfield to determine the number of bytes provided for the output buffer.
- For an externally-described file, use the *outputNullMap* pointer subfield to access the output null map for the file and use the *outputNullMapLen* subfield to determine the number of bytes provided for the output null map. The output null map is an array of indicators with one element for each field in the record. For a field which is null-capable, a value of '1' indicates that the field has the null value, and a value of '0' indicates that the field does not have the null value. For a field which is not null-capable, the value will always be '0'. If the *outputNullMap* pointer is null, the output null map is not relevant for the file within the RPG program.
- For an externally-described file, use the value of each field in the output buffer according to the data type of the field, and if null-values are relevant for the file, use the indicator in the output null map for each null-capable field.
- For a program-described file, use the output buffer according to the definition of the record that is agreed upon by the RPG programmer and the handler provider.

Handling output data using name-value information

- Use the *namesValues* pointer subfield to access the array of information about each field in the record format.
- For each *output* field, or each *input* or *output* field in the case of a subfile record format
  - Use the *value* pointer subfield to locate the buffer available for the field.
  - Use the *valueLenBytes* subfield to determine the the number of bytes available in the buffer.
  - Use the value of the field in the buffer, according to the data type of the field. See "Data types used in name-value information" on page 23 for more information.
  - If the *isNullCapable* subfield indicates that the field is null-capable, use the *hasNullValue* to determine whether the field has the null value.

Using search arguments

Several operations require the handler to use search arguments from the RPG program.

Using the relative record number

If the file is not keyed, search arguments are relative record numbers. Use the *rrn* subfield.

Handling key data using data structures

- Use the *key* pointer subfield to locate the key buffer for the file and use the *keyLen* subfield to determine the number of bytes provided for the key buffer.
- For an externally-described file, use the *keyNullMap* pointer subfield to access the key null map for the file and use the *keyNullMapLen* subfield to determine the number of bytes provided for the key null map. The key null map is an array of indicators with one element for each field in the key. For a key field which is null-capable, a value of '1' indicates that the field has the null value, and a value of '0' indicates that the field does not have the null value. For a key field which is not null-capable, the value will always be '0'. If the *keyNullMap* pointer is null, the key null map is not relevant for the file within the RPG program.
- For an externally-described file, use the value of each key field in the key buffer according to the data type of the field,and if null-values are relevant for the file, use the indicator in the key null map for each null-capable key field..
- For a program-described file, use the key buffer according to the definition of the key that is agreed upon by the RPG programmer and the handler provider.

Handling key data using name-value information

- Use the *keyNamesValues* pointer subfield to access the array of information about each field in the key.
- For each key field
  - Use the *value* pointer subfield to locate the buffer available for the key field.
  - Use the *valueLenBytes* subfield to determine the the number of bytes available in the buffer.
  - Use the value of the key field in the buffer, according to the data type of the field. See "Data types used in name-value information" on page 23 for more information.
  - If the *isNullCapable* subfield indicates that the key field is null-capable, use the *hasNullValue* to determine whether the key field has the null value.

Errors detected by the handler

When the handler detects an error, it must convey the error condition to the RPG program.

The handler may either send an exception message to its caller or it may set the *rpgStatus* subfield in the handler parameter to a valid RPG I/O status code.

The status code 1299 indicates a general I/O error, and can be used for any operation.

If the handler sends an exception message or if the handler fails with an unhandled exception, RPG will usually set the status code to 1299. For OPEN and CLOSE operations, RPG will set the status code to 1216 (error in explicit OPEN or CLOSE operation) or 1217 (error in implicit OPEN or CLOSE operation).

Tip: Sending an exception message is the recommended way to signal the error condition to the RPG program. The message will appear in the job log so the information in the message will be available after the program has completed. The handler can provide as much detail in the message as is needed to determine the reason for any unexpected errors.

Setting the feedback areas in the INFDS

The handler may optionally set feedback information that can be used to set the feedback areas in the RPG File Information Data Structure (INFDS) for the file.

This feedback information is not used directly by RPG or Open Access, but the RPG program may depend on it. For example, RPG programmers use the *rrn* subfield at position 397 of the INFDS for a DISK file to determine the relative record number of the current record. RPG programmers also use the *aid-byte* subfield at position 369 of the INFDS for a WORKSTN file to determine the function key that was pressed.

Tip: If the handler provides feedback information, it may not be necessary to set the entire feedback structure to match the values that would be set for a system file. RPG does not use the values in the feedback areas, so the handler only needs to provide the information needed by the specific RPG program.

Setting the feedback information

Each feedback area has an associated pointer and length subfield in the handler parameter.

- The handler must set the pointer subfield to storage that will remain active after the handler returns. It can be any type of storage other than automatic storage.
- The handler must set the length subfield to indicate the number of bytes of the storage that RPG should use to set the relevant section in the INFDS.

Open feedback
The open feedback section of the INFDS starts at position 81 and has a length of 160. Use the *openFeedback* pointer subfield and the *openFeedbackLen* length subfield.

I/O feedback
The I/O feedback section of the INFDS starts at position 241 and has a length of 126. Use the *ioFeedback* pointer subfield and the *ioFeedbackLen* length subfield.

Device-specific feedback

The device-specific feedback section of the INFDS starts at position 367. The length depends on the nature of the file. Use the *deviceFeedback* pointer subfield and the *deviceFeedbackLen* length subfield.

The RPG operations to be handled

The operation to be performed by the handler is specified by the *rpgOperation* subfield.

Each operation has a description of the operation's meaning in terms of RPG operation codes, and the actions that the handler must perform to fulfill the operation.

The name of the constant that defines the operation is followed by the numeric value of the constant in parentheses. This numeric value is provided for debugging purposes only; handler providers should use the named constant within their code.

QrnOperation_CHAIN (9)

RPG operation	Handler action
• If the *keyedFile* subfield is '0', the operation is random retrieval from a file by relative record number. • Otherwise, the operation is random retrieval from a file by key.	1. Locate the record specified by the search argument. 2. If the record is available, provide the data and set the *found* subfield to '1'. Otherwise, set the *found* subfield to '0'. 3. See “Using search arguments” on page 7for more information on using search arguments. 4. See “Handling input data” on page 5for more information on providing input data. **Remember:** The input data for a subfile record contains both input and output fields. The handler must retain the values of the output fields when the record is written so that it can provide the values of the output fields for subsequent input operations.

QrnOperation_CLOSE (18)

RPG operation	Handler action
Implicit or explicit close of the file. See “When the Open-Access file is closed” on page 14 for information on when a file is closed by RPG.	End the interaction with the resource or device.

QrnOperation_DELETE (16)

RPG operation	Handler action
Delete record by relative record number or key	Delete the record specified by the search argument. See “Using search arguments” on page 7for more information on using search arguments. **Note:** See QrnOperation_DELETE_CURRENT for a similar operation with no search argument specified.

QrnOperation_DELETE_CURRENT (19)

RPG operation	Handler action
Delete the current record	Delete the most recently input record if it is locked, or set the status to a failing value if it is not locked. **Note:** See QrnOperation_DELETE for a similar operation with a search argument specified.

QrnOperation_EXFMT (10)

RPG operation	Handler action
Write and read a record format from a user-interface file	1. Present the output data to the user including any subfile records associated with the record format 2. Receive the input data from the user, including input data for any subfile records associated with the record format 3. Provide the input data 4. If there are any subfile records associated with the record format • Retain an indication of whether each record was changed by the user. • Retain the subfile data for further QrnOperation_READC or QrnOperation_CHAIN for the subfile. 5. See "Handling input data" on page 5for more information on providing input data. 6. See "Handling output data" on page 6for more information on using output data.

QrnOperation_FEOD (17)

RPG operation	Handler action
Force end of data	Set the end-of-file condition

QrnOperation_OPEN (1)

RPG operation	Handler action
Implicit or explicit OPEN operation	Initialize the interaction with the resource or device.

QrnOperation_POSITION_END (3)

RPG operation	Handler action
SETLL *END	Set the file cursor to the end of the file.

QrnOperation_POSITION_START (2)

RPG operation	Handler action
SETLL *START	Set the file cursor to the beginning of the file.

QrnOperation_READ (4)

RPG operation	Handler action
• Implicit or explicit sequential input operation for a database file or record format. • Implicit or explicit input operation for a user-interface file or record format.	1. Move the file cursor forward to the next record. 2. Provide input data or set the *eof* subfield to '1'. 3. See “Handling input data” on page 5for more information on providing input data.

QrnOperation_READC (5)

RPG operation	Handler action
Read next changed subfile record	1. Move the file cursor forward to the next changed record. 2. Provide data or set the *eof* subfield to '1'. 3. See “Handling input data” on page 5for more information on providing input data. 1. **Remember:** The input data for a subfile record contains both input and output fields. The handler must retain the values of the output fields when the record is written so that it can provide the values of the output fields for subsequent input operations.

QrnOperation_READE (6)

RPG operation	Handler action
Read the next record if its key is equal to the search argument	1. Move the file cursor forward to the next record. 2. Provide input data if the key of the record matches the search argument, or set the *eof* subfield to '1'. 3. See “Using search arguments” on page 7for more information on using key data. 4. See “Handling input data” on page 5for more information on providing input data. **Note:** See QrnOperation_READPE_CURRENT for a similar operation with no search argument specified.

QrnOperation_READE_CURRENT (20)

RPG operation	Handler action
Read equal key, with no search argument specified	1. Move the file cursor forward to the next record. 2. Provide data if the record has the same key as the previously current record, or set the *eof* subfield to '1'. 3. See "Handling input data" on page 5for more information on providing input data. **Note:** See QrnOperation_READE for a similar operation with a search argument specified.

QrnOperation_READP (7)

RPG operation	Handler action
Sequential read previous operation	1. Move the file cursor backward to the previous record. 2. Provide input data or set the *eof* subfield to '1'. 3. See "Handling input data" on page 5for more information on providing input data.

QrnOperation_READPE (8)

RPG operation	Handler action
Read the previous record if its key is equal to the search argument	1. Move the file cursor backward to the previous record. 2. Provide input data if the key of the record matches the search argument, or set the *eof* subfield to '1'. 3. See "Using search arguments" on page 7for more information on using key data. 4. See "Handling input data" on page 5for more information on providing input data. **Note:** See QrnOperation_READE_CURRENT for a similar operation with no search argument specified.

QrnOperation_READPE_CURRENT (21)

RPG operation	Handler action
Read equal key previous, with no search argument specified	1. Move the file cursor backward to the previous record. 2. Provide data if the record has the same key as the previously current record, or set the *eof* subfield to '1'. 3. See "Handling input data" on page 5for more information on providing input data. **Note:** See QrnOperation_READPE for a similar operation with a search argument specified.

QrnOperation_SETGT (11)

RPG operation	Handler action
Set greater than	1. Set the file cursor to the first record greater than the search argument. 2. Set the *found* subfield to '1' if there is such a record. 3. See "Using search arguments" on page 7for more information on using search arguments.

QrnOperation_SETLL (12)

RPG operation	Handler action
Set lower limit	1. Set the file cursor to the last record less than or equal to the search argument. 2. Set the *found* subfield to '1' if there is such a record. 3. Set the *equal* subfield to '1' if the record is an exact match for the search argument. 4. See "Using search arguments" on page 7for more information on using search arguments.

QrnOperation_UNLOCK (13)

RPG operation	Handler action
Unlock record	Unlock the most recently input record if it is locked, or set the status to a failing value if it is not locked.

QrnOperation_UPDATE (14)

RPG operation	Handler action
Update the current record	Update the most recently input record if it is locked, or set the status to a failing value if it is not locked. See "Handling output data" on page 6for more information on using output data.

QrnOperation_WRITE (15)

RPG operation	Handler action
Write record	• If the record is a subfile record, retain the output data so it can be returned as part of the data on a subsequent QrnOperation_READC or QrnOperation_CHAIN. • If the record is not a subfile record, output the data to the device or resource. • See "Handling output data" on page 6for more information on using output data.

When the Open-Access file is closed

RPG closes the Open-Access file during normal RPG processing and during cancellation.

Implicit closing of Open-Access files

RPG attempts to close Open-Access files when the RPG module is no longer able to use the file.

- Global files in a cycle-main RPG module are implicitly closed when the main procedure ends with LR on, or when the main procedure ends abnormally.
- Automatic files in a subprocedure are implicitly closed when the subprocedure ends, normally or abnormally.
- Static files are implicitly closed, where possible, when the RPG module will no longer be active in the thread, activation group or job.

When the activation group ends

The RPG runtime uses the `CEE4RAGE()` built-in function to enable an activation group exit to run when the activation group ends. This function will call the Open-Access handler if the file is still open.

For a program or service program running in the default activation group, the activation group ends when the job ends.

Note: Calls to `CEE4RAGE()` are not permitted for programs compiled with DFTACTGRP(*YES). If an Open-Access file is still open when the job ends, the Open-Access handler will not be called.

When the secondary thread ends, for a module compiled with THREAD(*CONCURRENT)

The RPG runtime uses the *pthread* APIs `pthread_key_create()` and `pthread_set_specific()` to enable a *destructor* function to run when the thread ends. This function will call the Open-Access handler if the file is still open.

Note: The implementation of *pthreads* on the system might not call destructors when the thread ends using a mechanism outside of *pthreads*. One example is a thread ending due to an unhandled exception. Please see the *pthread* documentation in the Information Center for details.

Handler parameter

The same parameter is passed to the handler for all operations to the file.

The parameter is a data structure with several types of subfields:

Subfields that must be set by the handler for later use by RPG

The handler is responsible for setting some of the subfields in the handler parameter before returning to the RPG program. These subfields provide the information that tells RPG about the result of the operation. For example

- For the search operation QrnOperation_CHAIN, the *found* subfield must be set to '1' to indicate that a matching record was found.
- For an input capable operation, the handler must provide the data for all of the input capable fields.

Subfields that are set by RPG for use by the handler

RPG sets some subfields that tell the handler about the nature of the file, or the nature of the operation that the handler is required to perform. For example

- The *rpgOperation* subfield indicates which operation to perform.
- The *rpgDevice* subfield indicates the type of file defined in the RPG program.

Subfields that may be set by the handler for use by the RPG programmer

Some subfields provide feedback information that may be required by the RPG program. For example

- If the RPG program uses the **INKx* indicators, the *functionKey* subfield can be set to indicate that one of the function keys F1-F24 was pressed. If the RPG program monitors for status codes

1121-1126, the *functionKey* subfield can be set to indicate that one of the special keys such as the PRINT key was pressed.

- If the RPG program has a File Information Data Structure (INFDS) defined for the file, and the INFDS is longer than 80 bytes, then the handler may need to provide feedback for the file using the *openFeedback*, *ioFeedback*, and *deviceFeedback* subfields.

Subfields that allow the handler to maintain private information

- If the handler needs to share additional information with the RPG programmer, the second parameter of the HANDLER can be used in the RPG program to define a variable whose address will be passed to the handler in the *userArea* subfield of the handler parameter.
- If the handler needs to maintain private information across calls to the handler, the *stateInfo* subfield can be used to hold a pointer to the information.

The subfields of the main parameter structure

The subfields of the main parameter structure

The name of the main parameter structure is QrnOpenAccess_T.

Table 1. Subfields of QrnOpenAccess_T

Subfield	**Type**	**Set by**	**Used by**
structLen	UINT4	RPG	Handler
parameterFormat	CHAR(8)	RPG	Handler
userArea	Pointer [1]	RPG	Handler and RPG programmer
stateInfo	Pointer [3]	Handler	Handler
recordLevels [4]	Pointer [1]	RPG	Handler
inputBuffer	Pointer [1, 2]	Handler	RPG
inputNullMap [4]	Pointer [1, 2]	Handler	RPG
outputBuffer	Pointer [1, 2]	Handler	RPG
outputNullMap [4]	Pointer [1, 2]	Handler	RPG
namesValues [4]	Pointer [1, 2]	RPG and Handler	RPG and Handler
key	Pointer [1, 2]	Handler	RPG
keyNullMap [4]	Pointer [1, 2]	Handler	RPG
keyNamesValues [4]	Pointer [1, 2]	Handler	RPG
indara	Pointer [1, 2]	Handler	RPG
prtctl	Pointer [1, 2]	Handler	RPG
openFeedback	Pointer [3]	Handler	RPG
ioFeedback	Pointer [3]	Handler	RPG
deviceFeedback	Pointer [3]	Handler	RPG
externalFile	"System object structure, QrnObject_T" on page 25	RPG	Handler
externalMember	CHAR(10)	RPG	Handler

Table 1. Subfields of QrnOpenAccess_T (continued)

Subfield	**Type**	**Set by**	**Used by**
compileFile [4]	"System object structure, QrnObject_T" on page 25	RPG	Handler
recordName	CHAR(10)	RPG and Handler	RPG and Handler
rpgOperation	UINT4	RPG	Handler
rpgStatus	INT4	Handler	RPG
inputBufferLen	UINT4	RPG	Handler
inputNullMapLen [4]	UINT4	RPG	Handler
outputBufferLen	UINT4	RPG	Handler
outputNullMapLen [4]	UINT4	RPG	Handler
keyLen	UINT4	RPG	Handler
keyNullMapLen [4]	UINT4	RPG	Handler
inputDataLen	UINT4	Handler	RPG
openFeedbackLen	UINT4	Handler	RPG
ioFeedbackLen	UINT4	Handler	RPG
deviceFeedbackLen	UINT4	Handler	RPG
numKeys [4]	UINT4	RPG	Handler
rrn	UINT4	RPG and Handler	RPG and Handler
formLen	UINT4	RPG	Handler
formOfl	UINT4	RPG	Handler
sln	UINT4	RPG	Handler
alphaCcsids	UINT4	RPG	Handler
functionKey	UINT1	Handler	RPG
externallyDescribed	Indicator	RPG	Handler
keyedFile	Indicator	RPG	Handler
blocked	Indicator	RPG	Handler
eof	Indicator	Handler	RPG
found	Indicator	Handler	RPG
equal	Indicator	Handler	RPG
printerOverflow	Indicator	Handler	RPG
inputWithLock	Indicator	RPG	Handler
useNamesValues [4]	Indicator	Handler	RPG
isSubfile [4]	Indicator	RPG	Handler
canHandleCcsids [4]	Indicator	Handler	RPG
commit	Indicator	RPG	Handler

Table 1. Subfields of QrnOpenAccess_T (continued)

Subfield	Type	Set by	Used by
rpgDevice	CHAR(1)	RPG	Handler

Additional notes on the table

1. The "Set by" and "Used by" columns refer to the data that the pointer is pointing to.
2. The pointer is set by RPG. The handler sets the data pointed to by the pointer, usually with a variable or data structure based on the pointer.
3. The pointer is set by the handler.
4. This subfield is meaningful only for externally-described files.

Descriptions of the subfields

alphaCcsids

Whether alphanumeric fields have the job CCSID or the CCSID of the fields in the externally-described file. See "Constants QrnCcsids_*" on page 28.

Note: If this subfield has the value *QrnCcsids_FILE* indicating that alphanumeric fields have the CCSID of the fields in the file, the handler must set the *canHandleCcsids* subfield to '1'.

blocked

'1' if the file is defined to be blocked in the RPG program. '0' otherwise.

Note: This value is provided for information only. The Open-Access interface does not work with blocks of record, but the handler may use this information to control whether it block the records from the device or resource that it is dealing with.

canHandleCcsids

Set to '1' by the handler during the OPEN operation if the handler is able to handle alphanumeric fields with CCSIDs other than the job CCSID.

commit

'1' if the file should be opened under commitment control. '0' otherwise.

compileFile

The library and file that the RPG compiler used at compile-time for the description of an externally-described file.

Note: If an override was in effect for the file at compile-time, this reflects the actual file used at compile-time. For example, if the RPG file specification or EXTDESC keyword specifies file `MYFILE`, and there is an override for `MYFILE` to `MYLIB/OTHERFILE`, then the compileFile subfield will reflect library `MYLIB` and file `OTHERFILE`.

deviceFeedback

A pointer to the information that the handler provides to RPG to set the device-specific feedback part of the file's File Information Data Structure (INFDS). If this information is not supplied by the handler, the device-specific-feedback part of the INFDS is not updated by RPG. The length is specified by *deviceFeedbackLen*.

See "Setting the feedback areas in the INFDS" on page 8 for more information.

deviceFeedbackLen

The length of the device-specific-feedback information provided by the handler.

eof

Set to '1' by the handler if the file reached end of file for a sequential input operation or a WRITE operation to a subfile record. The handler can leave it at '0' otherwise to indicate that the record was successfully read or written.

equal
Set to '1' by the handler if an exact match was found by a SETLL operation. The handler can leave it at '0' otherwise.

externalFile
The library and file that the RPG program is opening at run time. It is the value specified by the EXTFILE keyword for the file. If the EXTFILE keyword was not specified, the library is *LIBL and the file is the internal name of the file in the RPG program..

Note: If an override is in effect for the file at run-time, this does not reflect the overridden file. For example, if the file specified by the EXTFILE keyword is `MYLIB/MYFILE` and there is an override present at run time that overrides file `MYFILE` to `OTHERLIB/OTHERFILE`, *externalFile* subfield will reflect the RPG value `MYLIB/MYFILE`. If the handler is interested in overrides, the handler may use an API to determine whether there is an override for the file.

externalMember
The member that the RPG program is opening at run time. It is the value specified by the EXTMBR keyword for the file. If the EXTMBR keyword was not specified, it is *FIRST.

Note: If an override is in effect for the file at run-time, this does not reflect the overridden member.

externallyDescribed
'1' if the file is externally-described in the RPG program. '0' otherwise.

formLen
Printer form length. Only meaningful for a program-described PRINTER file.

formOfl
Printer form overflow. Only meaningful for a program-described PRINTER file.

found
Set to '1' by the handler if a record was found by a positioning operation (CHAIN, SETLL, SETGT). The handler can leave it at '0' if a record was not found.

functionKey
For a user-interface input-capable operation, a function key value of 1-24 causes an *INKx indicator to be set on in the RPG program. A value of 121-126 indicates one of the PRINT, ROLLUP, ROLLDOWN, CLEAR, HELP or HOME keys, and causes the RPG status to be set to 1121 - 1126. Any other value causes an I/O error with status 1299. See “Constants QrnFunctionKey_* defining the values for the functionKey subfield” on page 28.

indara
A pointer to the INDARA array of 99 indicators used for the WORKSTN or PRINTER I/O operation. NULL if the file is not defined to use INDARA or if the handler is using name-value information.

inputBuffer
A pointer to a data structure in the format of the input buffer for the file or record. The length is given by inputBufferLen. NULL if the handler is using name-value information or if the input buffer is irrelevant to the operation.

inputBufferLen
The length of the input buffer. Zero if the input buffer is not relevant, or if name-value information is being used.

inputDataLen
When the file is externally-described and the record format was not specified by the RPG program, this subfield must be set by the handler to indicate the length of the data provided in the input buffer.

Note: This is not necessary if name-value information is being used.

inputNullMap
A pointer to the null-byte map of the input buffer for the file or record. The length is given by inputNullMapLen.

inputNullMapLen
The length of the input null-byte map. Zero if the input null-byte map is not relevant, or if name-value information is being used.

inputWithLock
: '1' if the input operation is intended to lock the record for later update or delete. '0' otherwise.

ioFeedback
: A pointer to the information that the handler provides to RPG to set the I/O feedback part of the file's File Information Data Structure (INFDS). The handler must set this pointer to storage that can be accessed by RPG. If this information is not supplied by the handler, the I/O-feedback part of the INFDS is not updated by RPG. The length is specified by *ioFeedbackLen*.

  See "Setting the feedback areas in the INFDS" on page 8 for more information.

ioFeedbackLen
: The length of the I/O-feedback information provided by the handler.

isSubfile
: '1' if the record format used for the operation is a subfile record. '0' otherwise.

key
: A pointer to a data structure in the format of the key structure for the file or record. The length is given by keyLen. NULL if the handler is using name-value information, or if the operation does not involve keys.

keyedFile
: '1' if the file is keyed in the RPG program. '0' otherwise.

keyLen
: The length of the key buffer. Zero if the key buffer is not relevant, or if name-value information is being used.

keyNamesValues
: A pointer to the QrnNamesValues_T structure listing the names, types, and values of each subfield in the key structure. NULL if the handler is not using name-value information, or if the operation does not involve keys, or if the name-value mode is not available due to the RPG programmer having specified an input operation to a file name where the file has more than one record format.

keyNullMap
: A pointer to the null-byte map of the key information for the file or record. The length is given by keyNullMapLen. NULL if the handler is using name-value information or if the key is irrelevant to the operation.

keyNullMapLen
: The length of the key null-byte map. Zero if the key null-byte map is not relevant, or if name-value information is being used.

namesValues
: A pointer to the QrnNamesValues_T structure listing the names, types, and values of each subfield in the record. NULL if the handler is not using name-value information, or if the operation does not involve input or output, or if the name-value mode is not available due to the RPG programmer having specified an input operation to a file name where the file has more than one record format.

numKeys
: The number of keys for a keyed-operation to an externally-described file or format.

openFeedback
: A pointer to the information that the handler provides to RPG to set the open-feedback part of the file's File Information Data Structure (INFDS). The handler must set this pointer to storage that can be accessed by RPG. If this information is not supplied by the handler, the open-feedback part of the INFDS is not updated by RPG. The length is specified by *openFeedbackLen*.

  See "Setting the feedback areas in the INFDS" on page 8 for more information.

openFeedbackLen
: The length of the open-feedback information provided by the handler.

outputBuffer
: A pointer to a data structure in the format of the output buffer for the file or record. The length is given by outputBufferLen. NULL if the handler is using name-value information or if the output buffer is irrelevant to the operation.

outputBufferLen
The length of the output buffer. Zero if the output buffer is not relevant, or if name-value information is being used.

outputNullMap
A pointer to the null-byte map of the output buffer for the file or record. The length is given by outputNullMapLen. NULL if the handler is using name-value information or if the output buffer is irrelevant to the operation.

outputNullMapLen
The length of the output null-byte map. Zero if the output null-byte map is not relevant, or if name-value information is being used.

parameterFormat
The format of the parameter structure. This subfield is provided to allow for the possibility of other formats in the future. The RPG program may be able to request a different format; this subfield would indicate the format of the parameter. ROIO0100 is currently the only format.

printerOverflow
Set to '1' by the handler if overflow was detected on an output operation to a printer file. The handler can leave it at '0' otherwise.

prtctl
A pointer to the print-control structure, QrnPrtctl_T, for a PRINTER file. NULL if the file is not a PRINTER file.

recordLevels
A pointer to the QrnRecordLevels_T structure listing the compile-time record-level identifier for each record in the file that is used by the RPG module. NULL if the file is a program-described file.

recordName
RPG sets this to the record name if the current I/O operation specifies the record name or if the file has only one record format. Otherwise, RPG sets this to blank. When it is blank, and it is an input operation to an externally-described file, the handler must set this subfield to the name of the record format that was used. This subfield is only meaningful for I/O operations that work with data. It is blank for other operations such as the OPEN operation.

rpgOperation
The operation being performed by the RPG program. See "The RPG operations to be handled" on page 9 for the constants *QrnOperation_** that define the possible values. Usually, *rpgOperation* maps directly to an RPG operation code, but some operation codes have more than one possible *rpgOperation* value. For example, the SETLL RPG operation may map to QrnOperation_SETLL, QrnOperation_POSITION_START, or QrnOperation_POSITION_END.

rpgDevice
The device type of the file as defined in the RPG program. See "Constants QrnRpgDevice_* defining the RPG device types" on page 27.

rpgStatus
The handler may leave this subfield set to zero to indicate that the operation was successful. If the handler determines that the operation is not successful, and the handler does not want to send an exception message to signal the condition, the handler may set this subfield to an RPG status code to have RPG signal the exception associated with that status code. Status 1299 is the RPG status code for general I/O errors, and can be used to signal an error condition for all operations. If this subfield it set to a value that does not represent a valid RPG error status code, the resulting behavior of the RPG program is undefined. See "Errors detected by the handler" on page 8 for more information on how the handler can indicate that it discovered an error.

rrn
Provided by RPG as the relative record number for an output operation or the search argument for a keyed operation. Provided by the handler for an input operation to a database file or a subfile. The relative record number is available to the RPG programmer through the RECNO.

Note: The RPG programmer may also attempt to obtain the relative record number through the Device-Specific Feedback area in the File Information Data Structure (INFDS). To support this, the handler must use the *deviceFeedback* subfield.

sln
The starting line number set by the RPG program for a user-interface record, related to the SLNO(*VAR) keyword for a system display file. If the SLN keyword is not specified for the file in the RPG program, this subfield will have a value of zero.

stateInfo
A private area defined by the handler. The handler has complete control of this pointer. The pointer will retain its value across calls to the handler. Usually the handler will set this pointer when the file is open, and it will clean up any storage associated with this pointer when the file is closed.

Note: The pointer must be set to storage that will continue to exist after the handler has returned.

structLen
The length of the structure.

useNamesValues
Set to '1' by the handler if the handler uses the name-value information rather than the data structure information for I/O buffers and keys. The handler can leave it at '0' otherwise. This subfield should only be changed during the OPEN operation. The behavior is undefined when this subfield is changed after the file has been opened.

userArea
A pointer to a handler-defined variable owned by the RPG program.

If the handler wants to share additional information with the RPG programmer, the second parameter of the HANDLER keyword can be used on the File specification for the Open-Access file to specify a variable to be passed to the handler. This subfield is a pointer to the RPG program variable. The RPG variable can have any data type; usually it is a data structure. The RPG program and the handler must agree on the type.

The subfields of the names-values structure

The names-values structure is *QrnNamesValues_T*.

Table 2. Subfields of QrnNamesValues_T

Subfield	Type	Set by	Used by
num	UINT4	RPG	Handler
field	"The subfields of the name-value structure describing one field" on page 22	RPG	Handler

Descriptions of the subfields

field
The array of name-value information.

num
The number of fields described by the information.

The subfields of the name-value structure describing one field

The name-value structure describing one field is *QrnNameValue_T*.

Table 3. Subfields of QrnNameValue_T

Subfield	Type	Set by	Used by
externalName	CHAR(10)	RPG	Handler
datatype	UINT1	RPG	Handler
numericDefinedLen	UINT1	RPG	Handler
decimals	UINT1	RPG	Handler
dtzFormat	UINT1	RPG	Handler
dtSeparator	CHAR(1)	RPG	Handler
input	Indicator	RPG	Handler
output	Indicator	RPG	Handler
isNullCapable	Indicator	RPG	Handler
hasNullValue	Indicator	RPG and Handler	RPG and Handler
valueLenBytes	UINT4	RPG and Handler	RPG and Handler
valueMaxLenBytes	UINT4	RPG	Handler
valueCcsid	INT4	RPG	Handler
value	Pointer [1,2]	RPG and Handler	Handler

Additional notes on the table

1. The "Set by" and "Used by" columns refer to the data that the pointer is pointing to.
2. The pointer is set by RPG. The handler works with the data pointed to by the pointer, usually with a variable or data structure based on the pointer.

Descriptions of the subfields

datatype

The data type of the field. For more information see "Data types used in name-value information" on page 23.

decimals

The number of decimal places for a decimal field. It is only meaningful for decimal fields.

dtzFormat

The format of a date, time, or timestamp field. It is only meaningful for date, time, or timestamp fields. For more information see "Constants QrnDtzFormat_* defining date, time, and timestamp formats" on page 25.

dtSeparator

The separator for a date or time field. It is only meaningful for date or time fields.

externalName

The name of the field in the externally-described file.

hasNullValue

'1' indicates that the field has a null value, '0' indicates that the field does not have a null value. Set by RPG to indicate whether an output field or key field has a null value. Set by the handler to indicate whether an input field has a null value. Meaningful only when *isNullCapable* has a value of '1'.

input
A value of '1' indicates that the field is input capable. This subfield is mainly useful for the EXFMT operation.

isNullCapable
A value of '1' indicates that the field is null capable in the RPG program.

numericDefinedLen
The defined length of a numeric field. For a decimal type, packed, zoned, or binary, it is the total number of digits. For a float or integer type, it is the number of bytes. It is only meaningful for numeric fields.

output
A value of '1' indicates that the field is output capable. This subfield is mainly useful for the EXFMT operation.

value
A pointer to the value of the field in human-readable form. The length, in bytes, of the value is given by the *valueLenBytes* subfield. For numeric values, this is the same value that is provided by the %CHAR built-in function. For date, time, and timestamp values, this is the same value that is provided by the %CHAR built-in function in the same format used by the field in the file.

- UCS-2 data is in the CCSID of the field in the file.
- Alphanumeric data is either in the CCSID of the field in the file or in the job CCSID.
- DBCS data is in the CCSID of the field in the file.
- The data for all other types is in the job CCSID.

When providing an input value, the handler must set the value in a similar form, and the handler must set the *valueLenBytes* subfield to reflect the length, in bytes, of the data provided. The length of the data must not exceed the number of bytes indicated by the *valueMaxLenBytes* subfield.

For more information, see “Data types used in name-value information” on page 23.

valueCcsid
The CCSID of the data in the value pointer. A value of zero indicates that the data is in the job CCSID.

valueLenBytes
Indicates the number of bytes in the data pointed to by the value pointer. Set by RPG for an output field or a key field. Set by the handler for an input field. It must not be greater than the *valueMaxLenBytes* subfield.

valueMaxLenBytes
Indicates the number of bytes in the buffer pointed to by the *value* pointer.

Data types used in name-value information

Data types used in name-value information.

The name of the constant that defines the type is followed by the numeric value of the constant in parentheses. This numeric value is provided for debugging purposes only; handler providers should use the named constant within their code.

QrnDatatype_Alpha (1)
Fixed-length character string in the job CCSID. For input operations, if the value provided by the handler is shorter than the size of the field, the value will be padded on the right with single-byte blanks when the value is used to set the RPG field.

QrnDatatype_AlphaVarying (2)
Varying-length character string in the job CCSID.

QrnDatatype_Date (12)
Date. The value is in the job CCSID. The format of the field is given by the *dtzFormat* subfield and the separator is given by the *dtSeparator* subfield. The value for an output field or a key field is in the form returned by the %CHAR built-in function of RPG, for example "2025-02-28". The value required for an input field must be in the same format. The value is converted to a date value using the same processing as the %DATE RPG built-in function.

QrnDatatype_Dbcs (5)
Fixed-length DBCS string. The CCSID is indicated by the *valueCcsid* subfield. For input operations, if the value provided by the handler is shorter than the size of the field, the value will be padded on the right with double-byte blanks when the value is used to set the RPG field.

QrnDatatype_DbcsVarying (6)
Varying-length DBCS string. The CCSID is indicated by the *valueCcsid* subfield.

QrnDatatype_Decimal (8)
Decimal numeric, used for packed, zoned, and binary RPG fields. The value is in the job CCSID. The total number of digits is given by the *numericDefinedLen* subfield and the number of decimal places is given by the *decimals* subfield. The value for an output field or a key field is in the form returned by the %CHAR built-in function of RPG, for example "12.34" or "-5,67". The value required for an input field must be in a similar form. The length of the value is not required to match the length of the field; the value is converted to a packed, zoned, or binary value using the same processing as the %DECH RPG built-in function. For example, if the field has 7 digits and 1 decimal position, the value provided for an input operation could be "-1.76". The value placed in the RPG field would be -1.8.

The decimal point may be the period or the comma.

QrnDatatype_Float (11)
Float numeric. The value is in the job CCSID. The number of bytes of the field (4 or 8) is given by the *numericDefinedLen* subfield. The value for an output field or a key field is in the form returned by the %CHAR built-in function of RPG, for example "+1.2300000E+00". The value required for an input field must be in a similar form. The length of the value is not required to match the length of the field and is not required to have the exponent. The value is converted to a float value using the same processing as the %FLOAT RPG built-in function. For example, if the field has length 4, the value provided for an input operation could be "-1.76". The value placed in the RPG float field would be -1.76E00.

QrnDatatype_Indicator (7)
Single-byte character with the value '0' or '1', used as a Boolean type in RPG.

QrnDatatype_Integer (9)
Integer numeric. The value is in the job CCSID. The number of bytes of the field (1, 2, 4, or 8) is given by the *numericDefinedLen* subfield. The value for an output field or a key field is in the form returned by the %CHAR built-in function of RPG, for example "12" or "-5". The value required for an input field must be in a similar form. The length of the value is not required to match the length of the field; the value is converted to a integer value using the same processing as the %INTH RPG built-in function. For example, if the field has length 4 (defined as having 10 digits in RPG), the value provided for an input operation could be "-1.76". The value placed in the RPG integer field would be -2.

QrnDatatype_Time (13)
Time. The value is in the job CCSID. The format of the field is given by the *dtzFormat* subfield and the separator is given by the *dtSeparator* subfield. The value for an output field or a key field is in the form returned by the %CHAR built-in function of RPG, for example "23.30.01". The value required for an input field must be in the same format. The value is converted to a date value using the same processing as the %TIME RPG built-in function.

QrnDatatype_Timestamp (14)
Timestamp. The value is in the job CCSID. The value for an output field or a key field is in the form returned by the %CHAR built-in function of RPG, for example "2010-12-25-23.30.01.000000". The value required for an input field must be in the same format. The value is converted to a date value using the same processing as the %TIMESTAMP RPG built-in function.

QrnDatatype_Unicode (3)
Fixed-length UCS-2 string. The CCSID is indicated by the *valueCcsid* subfield. For input operations, if the value provided by the handler is shorter than the size of the field, the value will be padded on the right with UCS-2 blanks when the value is used to set the RPG field.

QrnDatatype_UnicodeVarying (4)
Varying-length UCS-2 string. The CCSID is indicated by the *valueCcsid* subfield.

QrnDatatype_Unsigned (10)
Unsigned integer numeric. The value is in the job CCSID. The number of bytes of the field (1, 2, 4, or 8) is given by the *numericDefinedLen* subfield. The value for an output field or a key field is in the form

returned by the %CHAR built-in function of RPG, for example "12" or "5". The value required for an input field must be in a similar form. The length of the value is not required to match the length of the field; the value is converted to a unsigned integer value using the same processing as the %UNSH RPG built-in function. For example, if the field has length 2 (defined as having 5 digits in RPG), the value provided for an input operation could be "1.76". The value placed in the RPG unsigned field would be 2.

Constants QrnDtzFormat_* defining date, time, and timestamp formats

Constants QrnDtzFormat_* defining date, time, and timestamp formats.

The name of the constant that defines the type is followed by the numeric value of the constant in parentheses. This numeric value is provided for debugging purposes only; handler providers should use the named constant within their code.

QrnDtzFormat_DMY (7)
*DMY. Valid for date (DD/MM/YY). The separator is given by the *dtSeparator* subfield.

QrnDtzFormat_EUR (3)
*EUR. Valid for date (DD.MM.YYYY) and time (HH.MM.SS).

QrnDtzFormat_HMS (9)
*HMS. Valid for time (HH:MM:SS). The separator is given by the *dtSeparator* subfield.

QrnDtzFormat_ISO (1)
*ISO. Valid for date (YYYY-MM-DD), time (HH.MM.SS), and timestamp (YYYY-MM-DD-HH.MM.SS.UUUUUU).

QrnDtzFormat_JIS (4)
*JIS. Valid for date (YYYY-MM-DD) and time (HH:MM:SS).

QrnDtzFormat_JUL (8)
*JUL. Valid for date (YY/DDD). The separator is given by the *dtSeparator* subfield.

QrnDtzFormat_MDY (6)
*MDY. Valid for date (MM/YY/DD). The separator is given by the *dtSeparator* subfield.

QrnDtzFormat_USA (2)
*USA. Valid for date (MM/DD/YYYY) and time (HH:MM PM).

QrnDtzFormat_YMD (5)
*YMD. Valid for date (YY/MM/DD). The separator is given by the *dtSeparator* subfield.

System object structure, QrnObject_T

Structure QrnObject_T locates a system object

The subfields of the structure

Table 4. Subfields of QrnObject_T

Subfield	Type
name	CHAR(10)
library	CHAR(10)

Descriptions of the subfields

library
The library of the system object

name
The name of the system object

Print-control structure, QrnPrtctl_T

Structure QrnPrtctl_T defines an RPG print-control structure

The subfields of the structure

Table 5. Subfields of QrnPrtctl_T

Subfield	Type	Set by	Used by
spaceBefore	ZONED(3,0)	RPG	Handler
spaceAfter	ZONED(3,0)	RPG	Handler
skipBefore	ZONED(3,0)	RPG	Handler
skipAfter	ZONED(3,0)	RPG	Handler
currLine	ZONED(3,0)	Handler	RPG

Descriptions of the subfields

currLine
The current line in the file.

spaceAfter
The number of lines to advance after printing the line.

spaceBefore
The number of lines to advance before printing the line.

skipAfter
The line to skip to after advancing to the next page, after printing the line.

skipBefore
The line to skip to after advancing to the next page, before printing the line.

Record-levels structure, QrnRecordLevels_T

Structure QrnRecordLevels_T describes the record levels for the file

The *recordLevels* subfield of the handler parameter gives the compile-time record levels of the record formats used by the RPG program.

The subfields of the QrnRecordLevels_T structure

Table 6. Subfields of QrnRecordLevels_T

Subfield	Type
num	UINT4
levels	QrnRecordLevel_T

Descriptions of the subfields

levels
The array of level information.

num
The number of array elements.

Record-level structure for one level, QrnRecordLevel_T
Structure QrnRecordLevel_T describes the record level of one record format used in the file

The subfields of the QrnRecordLevel_T structure

Table 7. Subfields of QrnRecordLevel_T

Subfield	Type	
record	CHAR(10)	
level	CHAR(13)	

Descriptions of the subfields

level
The format level identifier for the record format at the time the RPG module was compiled. If the exact format of the record is important to the handler, the handler may call an API to retrieve the current format level identifier for the record format to verify that it matches the compile-time version of the record format.

record
The name of the record.

Data types used in the handler parameter

Data types used in the handler parameter.

CHAR(n)
Single-byte character string with **n** bytes

Indicator
Single-byte character with the value '0' or '1', used as a Boolean type in RPG

INT4
4-byte integer

UINT1
1-byte unsigned integer

UINT4
4-byte unsigned integer

ZONED(n,p)
Zoned decimal with **n** digits and **p** decimal places

Constants QrnRpgDevice_* defining the RPG device types

Constants QrnRpgDevice_* defining the RPG device types for the *rpgDevice* subfield.

The name of the constant that defines the device is followed by the value of the constant in parentheses. This value is provided for debugging purposes only; handler providers should use the named constant within their code.

These constants define the possible values for the *rpgDevice* subfield.

QrnRpgDevice_Database (D)
A database device, defined with device-type DISK in the RPG module.

QrnRpgDevice_Printer (P)
A printer device, defined with device-type PRINTER in the RPG module.

QrnRpgDevice_UserInterface (U)
A user-interface device, defined with device-type WORKSTN in the RPG module.

Constants QrnCcsids_*

Constants QrnCcsids_* defining the possible values for the *alphaCcsids* subfield.

The name of the constant is followed by the value of the constant in parentheses. This value is provided for debugging purposes only; handler providers should use the named constant within their code.

These constants define the possible values for the *alphaCcsids* subfield.

QrnCcsids_JOB (0)
Alphanumeric fields have the job CCSID.

QrnCcsids_FILE (1)
Alphanumeric fields have the same CCSID as the fields in the file.

QrnCcsids_N_A (2)
Not applicable. There are no alphanumeric fields in the file.

Constants QrnFunctionKey_* defining the values for the functionKey subfield

Constants QrnFunctionKey_* defining the values for the *functionKey* subfield.

The name of the constant that defines the function key is followed by the numeric value of the constant in parentheses. This numeric value is provided for debugging purposes only; handler providers should use the named constant within their code.

These constants can be used to set the *functionKey* subfield of the handler parameter.

QrnFunctionKey_None (0)
No function key was pressed.

QrnFunctionKey_01 (1) - QrnFunctionKey_24 (24)
One of the function keys F1 - F24 was pressed.

QrnFunctionKey_CLEAR (124)
The CLEAR key was pressed.

QrnFunctionKey_HELP (125)
The HELP key was pressed.

QrnFunctionKey_HOME (126)
The HOME key was pressed.

QrnFunctionKey_PRINT (121)
The PRINT key was pressed.

QrnFunctionKey_ROLLDOWN (123)
The ROLLDOWN key was pressed.

QrnFunctionKey_ROLLUP (122)
The ROLLUP key was pressed.

Restrictions for an Open-Access file

There are several RPG restrictions for an Open Access file.

- Name-value information is not available for an input operation to a file name when the file has more than one record format. If the handler has requested name-value information for the file with more than one record format, and the RPG program does an input operation by file name, the request for name-value information will be ignored for that particular operation and data structure buffers will be passed instead.
- If a global Open-Access file has a handler in the same module as the file, the USROPN keyword must be specified.
- An Open-Access file cannot be passed as a file parameter.
- An Open-Access file cannot be defined as a TEMPLATE file.

- An Open-Access file cannot be related to another file by the LIKEFILE keyword, either as the parent or the child.
- An Open-Access file cannot be defined as a record-address file, or the file that is process a record-address file.
- An Open-Access file cannot be defined as a table file.
- An Open-Access file cannot be a program-described WORKSTN file.
- An Open-Access file cannot use PRTCTL(*COMPAT).
- An Open-Access file cannot be a multiple-device file. The multiple-device keywords MAXDEV, DEVID, SAVEDS, SAVEIND cannot be used, and the POST operation for the file cannot have Factor 1 specified.

Writing a parser for the RPG DATA-INTO operation code

The DATA-INTO operation imports the data from a structured document into an RPG data structure. The operation requires a parser that parses the data in the document, and uses callback functions to pass the information about named values in the document to the RPG runtime, which places the data into an RPG data structure.

The parser may not always run to completion

Usually, the parser will be able to complete all its calls to the callback functions and return normally. However, in some cases, the call to the parser will be terminated before it has finished parsing. This can occur in the following circumstances:

- Sufficient data has been supplied to set the RPG variable.
- An error occurs while data is assigned to the RPG variable.
- The sequence of calls from the parser is not correct.

Tip: If your parser has code that needs to run when the parsing is complete, enable a cancel handler to perform this code. For example, if the parser is written in RPG, code the final code in an ON-EXIT section for the parser procedure, to ensure that it runs whether the parser ends normally or abnormally.

Parameter passed to a DATA-INTO parser

The parameter passed to a DATA-INTO parser is a data structure with type *QrnDiParm_T*. It has the following subfields.

data

A pointer to the data in the document. If the document was specified as a file name, this pointer contains the data in the file.

The CCSID of the data is indicated by the *dataCcsid* subfield.

The length of the data, in bytes, is indicated by the *dataLen* subfield.

env

A pointer to a data structure with type *QrnDiEnv_T*. This data structure has the procedure pointers for the callback procedures. See "Callback procedures for DATA-INTO parsers" on page 30.

handle

This subfield must be passed as the first parameter for every call to a callback procedure.

userParm

A pointer to either an RPG variable or to a null-terminated string. The *userParmisNullTermString* subfield indicates which it is.

If *userParmisNullTermString* does not have the value '1', no information is available about the data type of the variable. The RPG programmer coding the DATA-INTO operation is expected to be aware of the type of parameter that the parser supports.

Note: Do not modify the data pointed to by this pointer if the data is a null-terminated string.

dataLen
The length of the data pointed to by the *data* subfield, measured in bytes.

dataCcsid
The CCSID pointed to by the *data* subfield, measured in bytes. A value of zero indicates that the data is in the CCSID of the job.

userParmIsNullTermString
An RPG indicator (one-byte character) describing the data pointed to by the userParm subfield. A value of '1' indicates that the data is a null-terminated string. A value of '0' indicates that the data is an RPG variable.

Note: For parsers written in ILE RPG, the %PARMS built-in function cannot be used reliably if the parser is a procedure. However, since there is always exactly one parameter passed to a parser procedure, this should not be a concern.

Callback procedures for DATA-INTO parsers

The procedure pointers for the callback procedures are subfields of the parameter passed to the parser. See "Parameter passed to a DATA-INTO parser" on page 29.

In RPG, the callback procedures are enabled by setting the *env* pointer to the *env* subfield of the parameter passed to the parser. (1)

The prototypes for the callback procedures are in the QRNDTAINTO member of the QOAR/QRPGLESRC source file.

```
/COPY QOAR/QRPGLESRC,QRNDTAINTO
DCL-PI *n;
   parm LIKEDS(QrnDiParm_T) CONST;
END-PI;

env = parm.env; //  1
QrnDiStart (...);
```

In C, the callback procedure pointers are accessed as subfields of the *env* subfield of the parameter passed to the parser. 2

```
#include "QOAR/H,QRNDTAINTO"
main (int argc, void *argv[]) {
   const QrnDiParm_T *parm = (QrnDiParm_T *) argv[1];

   parm->env->QrnDiStart (...); //  2
}
```

The callback procedures

The first parameter for all the callback procedures is the *handle* subfield of the parameter that is passed to the parser.

QrnDiStart(handle)
This procedure must be called before any other callback procedures are called.

QrnDiFinish(handle)
This procedure must be called when parsing is complete.

QrnDiReportError(handle, errorCode, bytesParsed)
The parser can call this procedure to report that it found an error in the document.

Note: Control will not return to the parser after this procedure is called.

- *errorCode* must be greater than zero. The meaning of the error code is determined by the parser.

- *bytesParsed* indicates the number of bytes that the parser had parsed when it discovered the error.

QrnDiTrace(handle, msg, nested)
The parser can call this procedure to add its own tracing information. If the *nested* parameter has the value '1', the message will be issued at the current nesting level of the trace output. If the *nested* parameter has the value '0', the trace message will be issued starting in column 1 of the trace output.

Note: Tracing is enabled using the QIBM_RPG_DATA_INTO_TRACE_PARSER environment variable. See the DATA-INTO section of the IBM® Rational Development Studio for i: ILE RPG Reference.

QrnDiReportName(handle, name, nameLength)
The parser calls this procedure to report that it found a name in the document. This may be the name of a structure, an array, or some data.

- *name* is a pointer to the value of the name. The value must be in the same CCSID as the document. If the value is not in the same CCSID as the document, the `QrnDiReportNameCcsid` procedure must be used instead.

  The call to `QrnDiReportName` must be followed by a call to `QrnDiStartStruct`, `QrnDiStartArray`, `QrnDiReportValue`, or `QrnDiReportValueCcsid`.
- *nameLength* is the length of the name measured in bytes.

QrnDiReportNameCcsid(handle, name, nameLength, ccsid)
The parser calls this procedure to report that it found a name in the document when the value of the name is in a different CCSID from the CCSID of the document. This may be the name of a structure, an array, or scalar data.

The call to `QrnDiReportNameCcsid` must be followed by a call to `QrnDiStartStruct`, `QrnDiStartArray`,, `QrnDiReportValue`, or `QrnDiReportValueCcsid`.

- *name* is a pointer to the value of the name. The value must be in the CCSID specified by the *ccsid* parameter.
- *nameLength* is the length of the name measured in bytes.
- *ccsid* is the CCSID of the value of the name.

QrnDiReportValue(handle, value, valueLength)
The parser calls this procedure to report that it found a scalar value in the document.

- *name* is a pointer to the value. The value must be in the same CCSID as the document. If the value is not in the same CCSID as the document, the `QrnDiReportValueCcsid` procedure must be used instead.
- *valueLength* is the length of the value measured in bytes.

Normally, the name for the value must have been previously reported by a prior call to `QrnDiReportName`. However, the parser may call this procedure without previously reporting a name if the value is the only thing found in the document. In that case, RPG will assume that the name of the value is the same as the name of the target variable.

QrnDiReportValueCcsid(handle, value, valueLength, ccsid)
The parser calls this procedure to report that it found a value in the document when the value is in a different CCSID from the CCSID of the document.

- *name* is a pointer to the value. The value must be in the CCSID specified by the *ccsid* parameter.
- *valueLength* is the length of the value measured in bytes.
- *ccsid* is the CCSID of the value.

Normally, the name for the value must have been previously reported by a prior call to `QrnDiReportName`. However, the parser may call this procedure without previously reporting a name if the value is the only thing found in the document. In that case, RPG will assume that the name of the value is the same as the name of the target variable.

QrnDiReportAttr(handle, name, nameLength, value, valueLength)

The parser calls this procedure to report an attribute of a name. If the name also contains other child items, then they must be preceded by a call to `QrnDiStartStruct`.

For example, the following sequence of calls is valid, although not all the parameters are shown. The RPG variable matching "employee" must be a data structure with a subfield "type".

```
QrnDiReportName ("employee")
QrnDiReportAttr ("type", "manager")
```

The following sequence of calls is also valid, although not all the parameters are shown. The RPG variable matching "employee" must be a data structure with subfields "type" and "id".

```
QrnDiReportName ("employee")
QrnDiReportAttr ("type", "manager")
QrnDiStartStruct ()
   QrnDiReportName ("id")
   QrnDiReportValue ("12345");
QrnDiEndStruct ()
```

RPG supports two special attributes that give additional information about scalar fields.

fmt

This specifies the format for a data, time or timestamp value. The valid values for the follow the same rules as RPG uses for %DATE, %TIME and %TIMESTAMP respectively. The format defaults to *ISO with separators. The format may begin with an asterisk, and for formats that allow more than one separator, the format may be followed by the separator.

For example, the following values are valid for a "fmt" attribute: "dmy", "DMY", "*DMY/", "dmy-", "*DMY&", "dmy0".

adjust

A value of "right" causes the value for the name to be right-adjusted in the target RPG variable. Data is left-adjusted by default, but a value of "left" is also supported.

For example, the following sequence of calls is valid, although not all the parameters are shown. The RPG variable matching "id" must be a data structure with subfields "fmt" and "value".

```
QrnDiReportName ("id")
QrnDiReportAttr ("fmt", "plain")
QrnDiReportAttr ("value", "12A")
```

If the special attribute is not relevant, or the value is not valid, the attribute is treated as a normal attribute.

QrnDiReportAttrCcsid(handle, name, nameLength, nameCcsid, value, valueLength, valueCcsid)

The parser calls this procedure when the name or value are in a different CCSID from the CCSID of the document. See the discussion for `QrnDiReportAttr`.

QrnDiStartStruct(handle)

The parser calls this procedure to indicate that the previously reported name is a structure.

The parser may also call this procedure without previously reporting a name if the structure is the outermost stucture in the document. In that case, RPG will assume that the name of the structure is the same as the name of the target variable.

QrnDiEndStruct(handle)

The parser calls this procedure to report that the previously reported structure has ended.

QrnDiStartArray(handle)

The parser calls this procedure to indicate that the previously reported name is an array.

The parser may also call this procedure without previously reporting a name if the array is the outermost stucture in the document. In that case, RPG will assume that the name of the array is the same as the name of the target variable.

Note: The parser only reports the name of the array once. It reports each array element the same way as it reports non-array values, structures, or arrays. For example, if it is an array of scalar values, it calls `QrnDiReportName` to report the name of the array, then it calls `QrnDiStartStruct`, then it makes repeated calls to `QrnDiReportValue`, then it calls `QrnDiEndArray`.

QrnDiEndArray(handle)

The parser calls this procedure to report that the previously reported array has ended.

Tracing a DATA-INTO parser

If you want to know the sequence of calls from the parser, including the names and values, use the QIBM_RPG_DATA_INTO_TRACE_PARSER environment variable to enable tracing. See the DATA-INTO section of the IBM Rational Development Studio for i: ILE RPG Reference.

Here is an example of a trace:

```
---------------- Start ---------------
ReportName:  'petInfo'
StartStruct
 ReportName:  'pets'
 StartArray
  StartStruct
   ReportName:  'name'
   ReportValue:  'Spot'
   ReportName:  'type'
   ReportValue:  'dog'
   ReportName:  'age'
   ReportValue:  '3'
  EndStruct
  StartStruct
   ReportName:  'name'
   ReportValue:  'Puff'
   ReportName:  'type'
   ReportValue:  'cat'
   ReportName:  'age'
   ReportValue:  '7'
  EndStruct
 EndArray
 ReportName:  'veterinarian'
 ReportValue:  'Dr Smith'
EndStruct
---------------- Finish --------------
```

Note: If the trace output does not show up immediately, or if it flashes too quickly to see, you can view the standard output by calling the following ILE RPG program. Compile the program with CRTBNDRPG.

```
**FREE
CTL-OPT ACTGRP(*NEW);
DCL-PR printf EXTPROC(*DCLCASE);
  p POINTER VALUE OPTIONS(*STRING : *NOPASS);
END-PR;
DCL-PR getchar INT(10) EXTPROC(*DCLCASE) END-PR;
DCL-C EOL x'15';

printf (EOL);
getchar ();
return;
```

The following C program will accomplish the same thing. Compile the program with CRTBNDC.

```
#include <stdio.h>
main()
{
   printf("\n");
   getchar();
}
```

Example of a DATA-INTO parser

Note: Detailed explanation is provided only for the aspects of the example that are related to the DATA-INTO operation.

In this example, a parser parses a properties file for the DATA-INTO operation, and an RPG program uses the parser to import data from the properties file into a data structure.

This parser works with UCS-2 data. That means that it can parse documents in any CCSID. The "ccsid" option for the DATA-INTO operation defaults to "ccsid=ucs2", so RPG programmers using this parser will not have to worry about coding the "ccsid" option.

CAUTION: If the parser expected its data in the job CCSID, then data might be lost if the document contained data that could not be converted to the job CCSID.

The following shows the RPG program that uses the DATA-INTO operation.

Note the following aspects of the program:

1. The data structures are defined with a subfield for each property expected in the "properties" file.
2. The "properties" file is specified in the first operand of the %DATA built-in function. Option "doc=file" indicates that the first operand is the name of a file. Option "allowextra=yes" allows the "properties" file to have additional properties.
3. The program that does the parsing is specified as the first operand of the %PARSER built-in function. See "Program to parse a "properties" file" on page 35 for the source for the program.
4. The second DATA-INTO operation parses properties in a string. The "doc=file" option is not specified.
5. The string "sep=;" is specified as the second operand of the %PARSER built-in function. The parser will receive this value as a null-terminated string. See "Main procedure" on page 35 to see how the parser handles this null-terminated string. See "A DATA-INTO parser that uses a data structure as a communication area" on page 39 for an example of a parser which uses a data structure to communicate between the parser and the program with the DATA-INTO operation.

```
DCL-DS props1 QUALIFIED; //  1 
   company VARCHAR(30);
   language VARCHAR(10);
   version VARCHAR(10);
END-DS;
DCL-DS props2 QUALIFIED; //  1 
   city VARCHAR(30);
   province VARCHAR(10);
END-DS;

DCL-S propString VARCHAR(50) INZ('city=Toronto;province=Ontario;');

DATA-INTO props1 %DATA(propfileName : 'doc=file allowextra=yes') //  2 
                 %PARSER('PARSPROP'); //  3 

DATA-INTO props2 %DATA(propString : 'allowextra=yes') //  4 
                 %PARSER('PARSPROP' : 'sep=;'); // //  5 
```

Program to parse a "properties" file

1. Copy in the file with the definition for the parameter passed to the parser and the prototypes for the callback procedures.
2. Define named constants for the error codes issued by this parser. Each parser can define its own error codes.
3. Define other constants and templates related to parsing in UCS-2.
4. Define a data structure template to hold the information about the parse.

```
**free
ctl-opt main(parsProp);
ctl-opt option(*srcstmt);

/copy qoar/qrpglesrc,qrndtainto  1

// Error codes for this parser 2
dcl-c ERROR_missing_equal1        1;
dcl-c ERROR_blankName2            2;
dcl-c ERROR_blankInName3          3;

// Constants related to working in UCS-2 3
dcl-c UCS2_CCSID 13488;
dcl-c UTF16_CCSID 1200;
dcl-c CR %ucs2(X'0D');
dcl-c LF %ucs2(X'15');
dcl-c CHAR_SIZE 2; // The size of a UCS-2 character
dcl-s oneChar_t UCS2(1);

dcl-ds parseInfo_t template qualified; // 4
   lineStartOffset int(10);
   lineLength int(10);
   equalOffset int(10);
   curOffset int(10);
   sep varUcs2(20);
end-ds;
```

Main procedure

1. The parser is passed a single parameter. See "Parameter passed to a DATA-INTO parser" on page 29.
2. This parser supports a null-terminated string as the option for the %PARSER built-in function of the DATA-INTO operation. This parser expects the value of the null-terminated string to begin with "sep=", followed by the value that separates each option in the data. If this option is not specified, this parser assumes that the data came from a stream file, and that the CR and or LF characters end each line.
3. This parser expects the data to be UCS-2 or UTF-16. If the RPG programmer specified option "ccsid=job", this parser sends an escape message which will cause the DATA-INTO operation to fail.
4. Enable access to the callback procedures.
5. `QrnDiStart` must be called first.
6. Call `QrnDiStartStruct` to indicate that the document is a structure. Reporting a name for the outermost structure is not required.
7. The `parse()` procedure will report the names and values within the document.
8. Call `QrnDiEndStruct` to indicate that the outer data structure has ended.
9. `QrnDiFinish` must be called last.

```
dcl-proc parsProp;
   dcl-pi *n extpgm;
      parm likeds(QrnDiParm_T) const; // 1
   end-pi;
   dcl-ds parseInfo likeds(parseInfo_t) inz;
   dcl-s userParm varchar(30);

   if  parm.dataCcsid <> UCS2_CCSID // 2
   and parm.dataCcsid <> UTF16_CCSID;
     //We can only parse if option "ccsid=ucs2" was specified!
     //Send an escape message in this case, since it's a user error
     signalException ('%DATA must have ccsid=ucs2 for this parser'
                    : %proc());
     // Control will not reach here
   endif;

   if parm.userParmIsNullTermString; // 2
      userParm = %str(parm.userParm);
      if  %len(userParm) > 4
      and %scan('sep=' : userParm) = 1;
         // The parameter starts with 'sep='
         // The separator is the remaining part of the parameter
         parseInfo.sep = %subst(userParm : 5);
      endif;
   endif;

   pQrnDiEnv = parm.env; // 4

   QrnDiStart (parm.handle); // 5

   QrnDiStartStruct (parm.handle); // 6

   // Parse the document
   parse (parm : parseInfo); // 7

   // End the outer structure
   QrnDiEndStruct (parm.handle); // 8

   // End the parse
   QrnDiFinish(parm.handle); // 9

on-exit;
   // Nothing to do here yet
end-proc;
```

parse() procedure

This procedure loops through the document, reporting one property for each line it finds in the document.

```
dcl-proc parse;
   dcl-pi *n extproc(*dclcase);
      parserParm likeds(QrnDiParm_T) const;
      parseInfo likeds(parseInfo_t);
   end-pi;

   dow findNextLine (parserParm : parseInfo) = *on;
      reportProperty (parserParm : parseInfo);
   enddo;
   return;
end-proc;
```

findNextLine() procedure

This procedure loops through the data until it finds the end of a line.

1. If the options indicated a separator string, the parser checks whether it has found the separator. If so, the line is complete, and the procedure returns.

2. If the options did not indicate a separator string, the parser checks whether it has found an end-of-line character, either CR (carriage-return) or LF (line-feed). If the procedure had found any prior data for the line, the procedure returns. Otherwise, it begins a new line without returning the blank line. This allows the document to have lines that end with both CR and LF.
3. If the document is not valid according to the rules of this parser, the parser calls the "halt() procedure" on page 38 to indicate that the document is invalid.

   Note: Control does not return to the parser after a call to the `halt()` procedure due to the fact that the `halt()` procedure calls the `QrnDiReportError` procedure, which causes the parse to end immediately.

```
dcl-proc findNextLine;
   dcl-pi *n ind extproc(*dclcase);
      parserParm likeds(QrnDiParm_T) const;
      parseInfo likeds(parseInfo_t);
   end-pi;
   dcl-s viewCur like(oneChar_T) based(pData);
   dcl-s viewNext like(oneChar_T) based(pDataNext);
   dcl-s viewSep ucs2(MAX_SEP) based(pData); // must use %SUBST
   dcl-s sep varUcs2(MAX_SEP);
   dcl-s sepSize int(10);

   parseInfo.lineStartOffset = parseInfo.curOffset;
   parseInfo.lineLength = 0;
   parseInfo.equalOffset = 0;
   sep = parseInfo.sep;
   sepSize = %len(sep) * CHAR_SIZE;

   pData = parserParm.data + parseInfo.curOffset;
   dow parseInfo.curOffset < parserParm.dataLen;
      if %len(sep) > 0; // The separator is a string  1
         if parseInfo.curOffset + sepSize <= parserParm.dataLen
         and %subst(viewSep : 1 : %len(sep)) = sep;
            parseInfo.curOffset += sepSize;
            return *on; // End of line
         endif;
      endif;

      parseInfo.curOffset += CHAR_SIZE;
      if %len(sep) = 0 and (viewCur = CR or viewCur = LF);  2
         if parseInfo.lineLength > 0;
            return *on; // The line was not empty
         else; // The previous line was empty, so start again
            parseInfo.lineStartOffset = parseInfo.curOffset;
            parseInfo.lineLength = 0;
            parseInfo.equalOffset = 0;
         endif;
      else;
         parseInfo.lineLength += CHAR_SIZE;
         if viewCur = '=';
            parseInfo.equalOffset = parseInfo.curOffset - CHAR_SIZE;
         elseif viewCur = ' ';
            if parseInfo.equalOffset = 0; //  3
               if parseInfo.lineLength = 0; // Completely blank name
                  halt (parserParm : parseInfo : ERROR_blankName2);
               else; // Blanks are not allowed before the equal sign
                  halt (parserParm : parseInfo : ERROR_blankInName3);
               endif;
            endif;
         endif;
      endif;
      pData += CHAR_SIZE;// Next character (curOffset already updated)
   enddo;
   return parseInfo.lineLength > 0; // *ON if the line is not empty
end-proc;
```

reportProperty() procedure

This procedure reports the property found on a line in the document.

1. If the line is not valid, the parser reports the error using the `QrnDiReportError` callback.

Note: Control does not return to the parser after a call to `QrnDiReportError`.

2. The callback `QrnDiReportName` is used to report the name of the property. The DATA-INTO operation will use this name to locate a subfield in the target data structure.
3. The callback `QrnDiReportValue` is used to report the value of the property. The DATA-INTO operation will assign this value to the subfield that was located by the call to `QrnDiReportName`.

```
dcl-proc reportProperty;
   dcl-pi *n extproc(*dclcase);
      parserParm likeds(QrnDiParm_T) const;
      parseInfo likeds(parseInfo_t);
   end-pi;
   dcl-s len int(10);

   if parseInfo.equalOffset = 0;
      halt (parserParm : parseInfo //  1
          : ERROR_missing_equal1);
   elseif parseInfo.equalOffset = parseInfo.lineStartOffset;
      halt (parserParm : parseInfo //  1
          : ERROR_blankName2);
   endif;

   // Report the name
   len = parseInfo.equalOffset - parseInfo.lineStartOffset;
   QrnDiReportName (parserParm.handle //  2
                  : parserParm.data + parseInfo.lineStartOffset
                  : len);

   // Report the value
   len = parseInfo.lineLength - (len + CHAR_SIZE);
   QrnDiReportValue (parserParm.handle //  3
                  : parserParm.data + parseInfo.equalOffset + CHAR_SIZE
                  : len);
   return;
end-proc;
```

halt() procedure

This procedure reports a parsing error.

1. The parser reports the error using the `QrnDiReportError` callback.
2. Control does not return to the parser after a call to `QrnDiReportError`.

```
dcl-proc halt;
   dcl-pi *n extproc(*dclcase);
      parserParm likeds(QrnDiParm_T) const;
      parseInfo likeds(parseInfo_T) const;
      errorCode int(10) value;
   end-pi;

   QrnDiReportError (parserParm.handle //  1
                   : errorCode
                   : parseInfo.curOffset - 1);
   // Control will not reach here after the call to QrnDiReportError  2

end-proc;
```

signalException() procedure

This procedure sends an escape message.

1. The message is sent to the main

```
dcl-proc signalException;
   dcl-pi *n;
      msg varchar(200) const;
      mainProcName varchar(200) const;
   end-pi;
   dcl-pr QMHSNDPM extpgm;
      msgId char(7) const;
      msgFile likeds(qualMsgf);
      msgData char(500) const;
      dataLen int(10) const;
      msgType char(10) const;
      stackEntry char(10) const;
      stackOffset int(10) const;
      msgKey char(4) const;
      errorCode likeds(errcode);
   end-pr;
   dcl-ds qualMsgf qualified;
      msgf char(10) inz('QCPFMSG');
      lib char(10) inz('*LIBL');
   end-ds;
   dcl-ds errCode qualified;
      bytesProvided int(10) inz(0); // issue exception if bad parms
      bytesAvailable int(10) inz(0);
   end-ds;
   dcl-s msgkey char(4);

   QMHSNDPM ('CPF9898' : qualMsgf : msg : %len(msg) : '*ESCAPE'
           : mainProcName : 0  // Send to our main procedure  1
           : msgkey : errCode);
   // Control will not return here after sending the escape message
end-proc;
```

A DATA-INTO parser that uses a data structure as a communication area

This is not a complete example. It is intended to show how the DATA-INTO operation code and the parser can share information using a data structure. In this case, the second operand of the %PARSER built-in function is a data structure. The *userParm* subfield in the parameter passed to the parser is a pointer to that data structure, and the *userParmIsNullTermString* subfield has the value '0'.

The previous example shows how the DATA-INTO operation and the parser share information using a null-terminated string. In that case, the second operand of the %PARSER built-in function is a character expression. The *userParm* subfield is a pointer to a null-terminated string with the value of the character expression, and the *userParmIsNullTermString* subfield has the value '1'.

When a parser expects a data structure to be coded as the second operand of the %PARSER built-in function, there should be a copy file containing a template for the data structure. Both the parser and the RPG programs with the DATA-INTO operations will use the copy file to ensure that they agree on the nature of the data structure used as a communication area between the RPG program and the parser.

Copy file COMMAREA_H with a data structure template

```
   DCL-DS commArea_T QUALIFIED TEMPLATE;
      sub1 VARCHAR(10);
      sub2 INT(10);
   END-DS;
```

Program with a DATA-INTO operation

1. The /COPY directive copies in the source file with the template for the data structure used to communicate with the parser.
2. The communication-area data structure is defined.

3. The subfields of the communication-area data structure are set.
4. The communication-area data structure is specified as the second parameter of the %PARSER built-in function of the DATA-INTO operation.

```
/COPY COMMAREA_H   1
dcl-ds commArea likeds(commArea_T) inz; //  2

dcl-ds info qualified;
   city varchar(50);
   state varchar(50);
end-ds;

commArea.sub1 = 'x'; //  2
commArea.sub2 = 5;

DATA-INTO info %DATA(document : 'doc=file case=any')
                 %PARSER('PARS2' : commArea); //  4
```

Parser program

1. The /COPY directive copies in the source file with the template for the data structure used to communicate with the parser.
2. The communication-area data structure is defined with the BASED keyword.
3. The basing pointer for the data structure is set from the *userParm* subfield of the parameter passed to the parser.
4. The parser can now work with the subfields of the data structure passed from the DATA-INTO operation.

```
CTL-OPT main(pars2);

/COPY COMMAREA_H   1

/COPY QOAR/QRPGLESRC,QRNDTAINTO

DCL-PROC pars2;
   DCL-PI *n EXTPGM;
      parserParm LIKEDS(QrnDiParm_T) CONST;
   END-PI;
   DCL-DS userParm LIKEDS(commArea_T) BASED(pUserParm); //  2

   pUserParm = parserParm.userArea;  //  3

   IF userParm.sub1 = *BLANK;  //  4

   ...

END-PROC;
```

Writing a generator for the RPG DATA-GEN operation code

The DATA-GEN operation generates a structured document from an RPG variable. The operation requires a generator that creates the document using the names and values repeatedly passed to it by DATA-GEN. The generator uses callback functions to pass the text for the document to the RPG runtime, which places the data into an RPG variable or a file in the Integrated File System.

Handling clean-up activities

Usually, the generator can perform its clean-up activities during an "end' event. However, in some cases, there may not be an "end" event. This can occur in the following circumstances:

- An error occurs during the DATA-GEN operation.
- A DATA-GEN *START operation begins a sequence of DATA-GEN operations, but no matching DATA-GEN *END operation is done before the RPG procedure ends. See "Sequences of DATA-GEN operations" on page 41.

Tip: If your generator has code that needs to run when the generation is complete, set the *doTerminateEvent* subfield in the parameter passed to the generator to '1'. If that indicator is on after the most recent call to the generator, the RPG runtime will call the generator with a *QrnDgEvent_12_Terminate* event.

Warning: During the *QrnDgEvent_12_Terminate* event, the generator cannot call the callback procedures. Attempting to do so will result in an exception.

Sequences of DATA-GEN operations

A sequence of DATA-GEN operations begins with DATA-GEN *START and ends with DATA-GEN *END. A DATA-GEN operation is considered to be part of a sequence if the %DATA built-in function for the operation specifies the same file in the first operand, and specifies the "doc=file" option. A DATA-GEN operation that is part of a sequence must specify the "output=continue" option.

```
   DATA-GEN *START %DATA(filename : 'doc=file') %GEN('MYPGM');
   DATA-GEN ds1 %DATA(filename : 'doc=file output=continue') %GEN('MYPGM');
   DATA-GEN *END %DATA('myfile.txt' : 'doc=file') %GEN('MYPGM');
```

The DATA-GEN operations in a sequence may be done in different procedures, but they must all be in the same activation group and thread.

However, when the procedure with the DATA-GEN *START operation ends, the sequence ends even if the DATA-GEN *END operation has not been done. If you want to be sure to get control when the sequence ends, enable the QrnDgEvent_12_Terminate event. You request this event by setting the doTerminateEvent subfield of the parameter passed to the generator to '1'.

The generatorState pointer is available to all the operations in the sequence

If you have placed a pointer in the *generatorState*, this pointer will be available to all the subsequent operations in the sequence.

Performing clean-up activities when an operation is part of a sequence

If you have activities that must be performed at the end of the generation, and the operation is part of a sequence, you should wait to perform the clean-up activities until the QrnDgEvent_02_EndMultiple event or the QrnDgEvent_12_Terminate event, if the doTerminateEvent subfield of the parameter passed to the generator has been set to '1'.

Parameter passed to a DATA-GEN generator

The parameter passed to a DATA-GEN generator is a data structure with type *QrnDgParm_T*.

The parameter is defined in member QRNDTAGEN of source file QRPGLESRC for ILE RPG, QCBLLESRC for ILE COBOL, and H for ILE C/C++.

Warning: It is not possible to check the number of parameters passed to a generator if it is a bound procedure. The value returned by the %PARMS built-in function for an ILE RPG procedure, or the PARMS intrinsic function for an ILE COBOL procedure, is not reliable. However, since there is always exactly one parameter passed to a generator procedure, it should not be necessary to check the number of parameters passed to the procedure.

The parameter has the following subfields.

generatorState

(Input and output) A pointer holding state information owned by the generator. The generator may place a pointer here, and the same pointer will be available for all subsequent calls to the generator until the operation ends.

If the operation is part of a sequence of DATA-GEN operations, started by DATA-GEN *START and ended by DATA-GEN *END, the pointer will be available for all subsequent calls to the generator in the entire sequence. See “Sequences of DATA-GEN operations” on page 41.

Use the doTerminateEvent subfield to ensure that the generator is called for the *QrnDgEvent_12_Terminate* event, so that it can reliably perform cleanup-operations such as deallocating the *generatorState* pointer.

env

(Input only) A pointer to a structure with callback procedure pointers. See “Enabling calls to the callback procedures” on page 47 for information on the way to use this pointer to enable calling the callback procedures.

handle

(Input only)A pointer passed to every callback procedure

userParm

(Input and output) A pointer to the second operand of %GEN, owned by the RPG program, unless the *userParmType* subfield has the value *QrnUserParmType_nullTerminatedString* which indicates that the user-parameter is a null- terminated string, in which case it must be considered read-only.

The *userParmSize* subfield has the size of the value.

The *userParmCcsid* subfield has the CCSID of the value.

userParmSize

(Input only) This subfield indicates the length in bytes of the user-parameter.

If it is a null-terminated string (*userParmType* is *QrnUserParmType_nullTerminatedString*), this is the length of the string.

If it is a varying-length string (*userParmType* begins with *QrnUserParmType_var*), this includes the two or four bytes of the varying-length prefix. See subfield *userParmType* for the type of the user-parameter.

userParmCcsid

(Input only) This subfield indicates the CCSID of the user-parameter.

This subfield is not relevant if the *userParm* subfield is null, or if the value of the *userParmType* subfield is *QrnUserParmType_notPassed*, *QrnUserParmType_dataStruct*, or *QrnUserParmType_other*. For the possible values of the *userParmType* subfield, see “Types of the user-parameter passed to the DATA-GEN generator” on page 46.

userParmType

(Input only) This subfield indicates the type of the value pointed to by the *userParm* pointer.

Note: Full type information is not available for every type of user-parameter.

For the possible values of this parameter, see “Types of the user-parameter passed to the DATA-GEN generator” on page 46.

outputIsToFile

(Input only) An indicator whose value is '1' if the output from the callbacks is being written to a stream file; '0' otherwise.

doTerminateEvent

(Output) An indicator that is set to '1' by the generator if DATA-GEN should call the generator with the *QrnDgEvent_12_Terminate* event so that it can do clean-up. The generator can change it to '0' if the *QrnDgEvent_12_Terminate* event is no longer needed.

The value of this indicator following the most recent call to the generator is used by DATA-GEN when deciding whether to call the generator with the *QrnDgEvent_12_Terminate* event.

isPartOfSequence
(Input only) An indicator whose value is '1' if the DATA-GEN operation is part of a sequence; '0' otherwise.

name
(Input only) The name of the array, data structure, or scalar value. The data type is defined as *QrnDgName_t*. It is a varying length UTF-16 value, with a maximum length of 4096.

This subfield can be used directly if the generator is written in RPG.

```
   if parm.name = *blanks;
   ...
```

For generators written in other languages, this subfield is a data structure with *len* and *name* subfields.

This value is blank for the following events:

- QrnDgEvent_01_StartMultiple
- QrnDgEvent_02_EndMultiple
- QrnDgEvent_03_Start
- QrnDgEvent_04_End
- QrnDgEvent_12_Terminate

event
(Input only) This subfield indicates the type of event that is passed to the generator. See "Events for DATA-GEN generators" on page 49.

***ds* (*u.ds* for generators written in C)**
(Input only) This subfield must only be used for the following events:

- QrnDgEvent_05_StartStruct
- QrnDgEvent_06_EndStruct

The type of the *ds* data structure is *QrnDgDs_T*. See "Data structure information (type QrnDgDs_T)" on page 43 for the description of this data structure.

***array* (*u.array* for generators written in C)**
(Input only) This subfield must only be used for the following events:

- QrnDgEvent_07_StartScalarArray
- QrnDgEvent_08_EndScalarArray
- QrnDgEvent_09_StartStructArray
- QrnDgEvent_10_EndStructArray

The type of the subfield is *QrnDgArray_T*. See "Array information (type QrnDgArray_T)" on page 44 for the description of this data structure.

***scalar* (*u.scalar* for generators written in C)**
(Input only) This subfield must only be used for the QrnDgEvent_11_ScalarValue event. The type of the subfield is *QrnDgScalar_T*. See "Scalar information (type QrnDgScalar_T)" on page 45 for the description of this data structure.

Data structure information (type QrnDgDs_T)

An RPG data structure or data structure subfield is described by the *ds* subfield of the parameter passed to the generator. The type of the *ds* subfield is *QrnDgDs_T*.

All the subfields of the *QrnDgDs_T* are input-only:

elem
If the data structure is part of an array, this subfield indicates the element number. Otherwise, this subfield is zero. The element number is 1-origin, beginning at 1 and ending with the number of elements indicated by the *totalElems* subfield.

totalElems
If the data structure is part of an array, this subfield indicates the number of elements in the array. Otherwise, this subfield is zero. The current element is indicated by the*elem* subfield.

numSubfields
This subfield indicates the number of subfields for this data structure.

Tip: If the names of the subfields are needed in advance, this subfield can be used together with the `QrnDgGetSubfieldName` callback to obtain the subfield names during the `QrnDgEvent_05_StartStruct` or `QrnDgEvent_06_EndStruct` events.

subfieldNumber
If this RPG data structure is a subfield of another RPG data structure, this subfield indicates the subfield number in that data structure. If this RPG data structure is the top-level variable for the DATA-GEN operation, this subfield is zero. The subfield number is 1-origin, beginning at 1 and ending with the number of subfields in the data structure.

isExtDesc
This subfield is an indicator. The value is '1' if the data structure is externally-described, and '0' otherwise.

extLibrary
If the data structure is externally-described, this subfield indicates the library with the external file. Otherwise, this subfield is blank.

extFile
If the data structure is externally-described, this subfield indicates the external file. Otherwise, this subfield is blank.

extFormat
If the data structure is externally-described, this subfield indicates the external record format. Otherwise, this subfield is blank.

recordLevelId
If the data structure is externally-described, this subfield indicates the level ID of the external record format. Otherwise, this subfield is blank.

Array information (type QrnDgArray_T)

An RPG data array or array subfield is described by the *array* subfield of the parameter passed to the generator. The type of the *array* data structure is *QrnDgArray_T*.

Each element of the array will be described by the QrnDgEvent_05_StartStruct event or the QrnDgEvent_11_ScalarValue event.

All the subfields of the *QrnDgArray_T* data structure are input-only:

totalElems
This subfield indicates the number of elements in the array.

numSubfields
If this is an array of data structures, this subfield indicates the number of subfields for this data structure. Otherwise, this subfield is zero.

Tip: If the names of the subfields are needed in advance, this subfield can be used together with the `QrnDgGetSubfieldName` callback to obtain the subfield names during the QrnDgEvent_09_StartStructArray or QrnDgEvent_10_EndStructArray events.

subfieldNumber
If this RPG array is a subfield of an RPG data structure, this subfield indicates the subfield number in that data structure. If this RPG array is the top-level variable for the DATA-GEN operation, this subfield

is zero. The subfield number is 1-origin, beginning at 1 and ending with the number of subfields in the data structure.

Scalar information (type QrnDgScalar_T)

Scalar information describes an RPG field, subfield, or array element that is not a data structure. The type of the *scalar* data structure is *QrnDgScalar_T*.

All the subfields of the *QrnDgScalar_T)* data structure are input-only:

elem
If the scalar item is part of an array, this subfield indicates the element number. Otherwise, this subfield is zero. The element number is 1-origin, beginning at 1 and ending with the number of elements indicated by the *totalElems* subfield.

totalElems
If the scalar item is part of an array, this subfield indicates the number of elements in the array. Otherwise, this subfield is zero. The current element is indicated by the*elem* subfield.

definedCcsid
This subfield indicates the CCSID of the RPG scalar item. If the data type of the RPG scalar item is not alphanumeric, UCS-2, or graphic, this subfield is zero.

Note: This is not the CCSID of the data in the *value* subfield.

subfieldNumber
If this RPG scalar value is a subfield of an RPG data structure, this subfield indicates the subfield number in that data structure. If this RPG scalar value is the top-level variable for the DATA-GEN operation, this subfield is zero. The subfield number is 1-origin, beginning at 1 and ending with the number of subfields in the data structure.

dataType
This subfield indicates the data type of the scalar value. See "Data types used in name-value information" on page 23.

Note: Varying-length string types are not distinguished from fixed-length string types.

- This subfield is QrnDatatype_Alpha for all alphanumeric scalar values.
- This subfield is QrnDatatype_Unicode for all UCS-2 scalar values.
- This subfield is QrnDatatype_Dbcs for all graphic scalar values.

dtzFormat
If the RPG scalar value is of type Date or Time, this subfield gives the format. See "Constants QrnDtzFormat_* defining date, time, and timestamp formats" on page 25.

separator
If the RPG scalar value is of type Date or Time, this subfield gives the separator character.

If the RPG scalar value is numeric, this subfield gives the decimal point, either period or comma. The decimal point character for numeric values is determined by the DECEDIT keyword in the Control statements of the RPG module. If the DATA-GEN operation is part of a sequence of operations, the decimal point may not be the same for all the DATA-GEN operations in the sequence. For example, a numeric value of 12.3 for one DATA-GEN operation in the sequence might be passed to the generator as "12,3", and the same numeric value for another DATA-GEN operation in a different module might be "12.3".

valueLenBytes
This subfield indicates the length of the value measured in bytes.

valueLenChars
This subfield indicates the length of the value measured in UTF-16 double-byte characters.

valueCcsid
This subfield indicates the CCSID of the *value* subfield.

Note: This is not necessarily the CCSID of the RPG scalar item. Use the *definedCcsid* to determine the CCSID of the RPG scalar item.

value

This subfield is a pointer to the UTF-16 value of the RPG scalar item, in human-readable form.

The length of the value in bytes is given by the *valueLenBytes* subfield.

The length of the value in double-byte characters is given by the *valueLenChars* subfield.

String data (types QrnDatatype_Alpha, QrnDatatype_Unicode, and QrnDatatype_Dbcs) is trimmed of leading and trailing blanks unless option "trim=none" is specified for the DATA-GEN operation. See the DATA-GEN section of the IBM Rational Development Studio for i: ILE RPG Reference for information on the *trim* option.

Numeric data is edited with a leading sign if the value is negative. For QrnDatatype_Decimal and QrnDatatype_Float data, the decimal point is either a period or a comma, determined by the DECEDIT keyword of the RPG module. See *scalar.separator*.

Date, Time, and Timestamp data is edited according to the format and separators of the field in the RPG module.

Types of the user-parameter passed to the DATA-GEN generator

The name of the constant that defines the type is followed by the character value of the constant in parentheses. This character value is provided for debugging purposes only; programmers should use the named constant within their code.

QrnUserParmType_notPassed ('0')

The user-parameter was not specified in the RPG program. The *userParm* subfield of the parameter passed to the generator is null.

QrnUserParmType_nullTerminatedString ('1')

The user-parameter is a null-terminated string. The length can be determined by locating the null-terminator, or by using the *userParmSize* subfield of the parameter passed to the generator.

The value is in the job ccsid.

QrnUserParmType_indicator ('2')

The user-parameter is an RPG indicator with a value of '0' indicating "false" or '1' indicating "true".

QrnUserParmType_char ('3')

The user parameter is a fixed-length alphanumeric string. See the *userParmSize* subfield for the length and the *userParmCcsid* subfield for the CCSID.

QrnUserParmType_varchar_2 ('4')

The user parameter is a varying-length alphanumeric string with a 2-byte varying-length prefix. See the *userParmSize* subfield for the total size in bytes of the variable, including the varying-length prefix, and the *userParmCcsid* subfield for the CCSID of the data in the string. The length in characters of the data in the string can be determined from the value of the varying-length prefix.

QrnUserParmType_varchar-4 ('5')

The user parameter is a varying-length alphanumeric string with a 4-byte varying-length prefix. See the *userParmSize* subfield for the total size in bytes of the variable, including the varying-length prefix, and the *userParmCcsid* subfield for the CCSID of the data in the string. The length in characters of the data in the string can be determined from the value of the varying-length prefix.

QrnUserParmType_graph ('6')

The user parameter is a fixed-length graphic string. See the *userParmSize* subfield for the length in bytes and the *userParmCcsid* subfield for the CCSID.

The length in double-byte graphic characters can be determined by dividing the length in bytes by 2.

QrnUserParmType_vargraph_2 ('7')
The user parameter is a varying-length graphic string with a 2-byte varying-length prefix. See the *userParmSize* subfield for the total size in bytes of the variable, including the varying-length prefix, and the *userParmCcsid* subfield for the CCSID of the data in the string.

The length in double-byte graphic characters of the data in the string can be determined from the value of the varying-length prefix.

QrnUserParmType_vargraph_4 ('8')
The user parameter is a varying-length graphic string with a 4-byte varying-length prefix. See the *userParmSize* subfield for the total size in bytes of the variable, including the varying-length prefix, and the *userParmCcsid* subfield for the CCSID of the data in the string.

The length in double-byte graphic characters of the data in the string can be determined from the value of the varying-length prefix.

QrnUserParmType_ucs2 ('9')
The user parameter is a fixed-length UCS-2 or UTF-16 string. See the *userParmSize* subfield for the length in bytes and the *userParmCcsid* subfield for the CCSID. If the CCSID is 13488, the value is UCS-2. If the CCSID is 1200, the value is UTF-16.

The length in double-byte characters can be determined by dividing the length in bytes by 2.

QrnUserParmType_varucs2_2 ('a')
The user parameter is a varying-length UCS-2 or UTF-16 string with a 2-byte varying-length prefix. See the *userParmSize* subfield for the total size in bytes of the variable, including the varying-length prefix, and the *userParmCcsid* subfield for the CCSID of the data in the string. If the CCSID is 13488, the value is UCS-2. If the CCSID is 1200, the value is UTF-16.

The length in double-byte characters of the data in the string can be determined from the value of the varying-length prefix.

QrnUserParmType_varucs2_4 ('b')
The user parameter is a varying-length UCS-2 or UTF-16 string with a 4-byte varying-length prefix. See the *userParmSize* subfield for the total size in bytes of the variable, including the varying-length prefix, and the *userParmCcsid* subfield for the CCSID of the data in the string. If the CCSID is 13488, the value is UCS-2. If the CCSID is 1200, the value is UTF-16.

The length in double-byte characters of the data in the string can be determined from the value of the varying-length prefix.

QrnUserParmType_dataStruct ('c')
The user parameter is a data structure. Information about the subfields is not available. See the *userParmSize* subfield for the total size in bytes of the data structure.

QrnUserParmType_other ('d')
The type of the user parameter is not available. See the *userParmSize* subfield for the total size in bytes of the value.

Callback procedures for DATA-GEN generators

The procedure pointers for the callback procedures are subfields of the parameter passed to the generator. See "Parameter passed to a DATA-GEN generator" on page 41.

Enabling calls to the callback procedures

In RPG, the callback procedures are enabled by setting the *env* pointer to the *env* subfield of the parameter passed to the generator. (1)

The prototypes for the callback procedures are in the `QRNDTAGEN` member of the `QOAR/QRPGLESRC` source file.

```
/COPY QOAR/QRPGLESRC,QRNDTAGEN
DCL-PI *n;
   parm LIKEDS(QrnDgParm_T);
END-PI;

env = parm.env; //  1
QrnDgAddText (...);
```

In C, the callback procedure pointers are accessed as subfields of the *env* subfield of the parameter passed to the generator. 2

```
#include "QOAR/H,QRNDTAGEN"
main (int argc, void *argv[]) {
   QrnDgParm_T *parm = (QrnDgParm_T *) argv[1];

   parm->env->QrnDgAddText (...); //  2
}
```

The callback procedures

The first parameter for all the callback procedures is the *handle* subfield of the parameter that is passed to the generator.

QrnDgReportError(handle, returnCode)
The generator can call this procedure to report that it found an error in the document. The generator determines the meanings of its own return codes.

```
QrnDgReportError (parm.handle : ERR_CODE_1_INVALID_NAME);
```

Note: Control will not return to the generator after this procedure is called.

QrnDgTrace(handle, message, nested)
The generator can call this procedure to add its own tracing information. If the *nested* parameter has the value '1', the message will be issued at the current nesting level of the trace output. If the *nested* parameter has the value '0', the trace message will be issued starting in column 1 of the trace output.

```
QrnDgTrace (parm.handle : 'Getting the subfield names' : '1');
```

Note: Tracing is enabled using the QIBM_RPG_DATA_GEN_TRACE environment variable. See the DATA-GEN section of the IBM Rational Development Studio for i: ILE RPG Reference.

QrnDgAddText(handle, text, textChars)
The generator can call this procedure to add UTF-16 text.

Note: The length is the number of double-byte characters.

```
CTL-OPT CCSID(*UCS2:1200);
...
DCL-S name VARUCS2(200);
...
QrnDgAddText (parm.handle
            : %ADDR(name:*DATA)
            : %LEN(name));
```

QrnDgAddTextCcsid(handle, text, textBytes, ccsid)

The generator can call this procedure to add text in any CCSID.

Note: The length is the number of bytes. If the data is a double-byte data type, the number of characters must be multiplied by 2 to obtain the number of bytes.

```
DCL-S name VARCHAR(200) CCSID(1208);
DCL-S addr VARUCS2(200) CCSID(13488);
...
QrnDgAddTextCcsid (parm.handle
                 : %ADDR(name:*DATA)
                 : %LEN(name)
                 : 1208);
QrnDgAddTextCcsid (parm.handle
                 : %ADDR(addr:*DATA)
                 : %LEN(addr) * 2
                 : 1208);
```

QrnDgAddTextString(handle, text)

The generator can call this procedure to add text that is in a null-terminated string in the job CCSID.

```
QrnDgAddTextString (parm.handle : 'hello');
```

QrnDgAddTextNewLine(handle)

The generator can call this procedure to add new-line characters to the output.

Tip: The generator may only want to add new-line characters to the output when the output is intended for a stream file. The *outputIsToFile* subfield of the parameter passed to the generator has a value of '1' if the output is intended for a stream file.

```
if parm.outputIsFile;
   QrnDgAddTextNewLine (parm.handle);
endif;
```

name = QrnDgGetSubfieldName(handle, index)

The generator can call this procedure to find the name of a subfield of the data structure during one of the events in the list below. The index is 1-origin, beginning at 1 and ending at the number of subfields. The number of subfields can be obtained using the subfield indicated after each event in the list.

- QrnDgEvent_05_StartStruct (parm.ds.numSubfields)
- QrnDgEvent_06_EndStruct (parm.ds.numSubfields)
- QrnDgEvent_09_StartStructArray (parm.array.numSubfields)
- QrnDgEvent_10_EndStructArray (parm.array.numSubfields)

if parm.event = QrnDgEvent_05_StartStruct or parm.event = QrnDgEvent_06_EndStruct; for i = 1 to parm.ds.numSubfields; name = QrnDgAddTextNewLine (parm.handle : i); endfor; elseif parm.event = QrnDgEvent_09_StartStructArray or parm.event = QrnDgEvent_10_EndStructArray; for i = 1 to parm.array.numSubfields; name = QrnDgAddTextNewLine (parm.handle : i); endfor; endif;

Events for DATA-GEN generators

The *event* subfield of the parameter passed to the DATA-GEN generator indicates the type of information that is passed to the generator.

QrnDgEvent_01_StartMultiple
This event indicates that it is a DATA-GEN *START operation. This operation begins a sequence of DATA-GEN operations that ends with a DATA-GEN *END operation. See "Sequences of DATA-GEN operations" on page 41.

QrnDgEvent_02_EndMultiple
This event indicates that it is a DATA-GEN *END operation. This operation ends the sequence of DATA-GEN operations that begins with a DATA-GEN *START operation. See "Sequences of DATA-GEN operations" on page 41.

QrnDgEvent_03_Start
This event is the first event for a DATA-GEN operation where the first operand is a variable.

QrnDgEvent_04_End
This event is the last event for a DATA-GEN operation where the first operand is a variable unless you have requested the QrnDgEvent_12_Terminate event using the doTerminateEvent subfield of the parameter passed to the generator.

QrnDgEvent_05_StartStruct (*parm.ds*)
This event indicates that the RPG variable or subfield is a data structure. Refer to the name subfield and ds subfields of the parameter passed to the generator.

The subfields of the RPG data structure are described in subsequent events.

QrnDgEvent_06_EndStruct (*parm.ds*)
This event indicates the end of the description of the subfields for the RPG data structure. Refer to the name subfield and ds subfields of the parameter passed to the generator.

QrnDgEvent_07_StartScalarArray (*parm.array*)
This event indicates that the RPG variable is an array of scalar items. The array elements are described in subsequent events. Refer to the name subfield and array subfields of the parameter passed to the generator.

QrnDgEvent_08_EndScalarArray (*parm.array*)
This event indicates the end of the description of the elements of the array. Refer to the name subfield and array

QrnDgEvent_09_StartStructArray (*parm.array*)
This event indicates that the RPG variable is an array of data structures. The array elements are described in subsequent events. Refer to the name subfield and array

QrnDgEvent_10_EndStructArray (*parm.array*)
This event indicates the end of the description of the elements of the array. Refer to the name subfield and array

QrnDgEvent_11_ScalarValue (*parm.scalar*)
This event indicates that the RPG variable is a scalar value. Refer to the name subfield and scalar

QrnDgEvent_12_Terminate
This event indicates that the generator can safely perform its clean-up activities. The generator is called with this event only if the generator has set the doTerminateEvent subfield of the parameter passed to the generator to a value of '1'.

If the DATA-GEN operation is part of a sequence of events started by DATA-GEN *START and ended by DATA-GEN *END, the generator is called with this event after the DATA-GEN *END has completed, or when the RPG runtime has determined that the DATA-GEN *END operation will never be done for this sequence. See "Sequences of DATA-GEN operations" on page 41.

If the DATA-GEN operation is not part of a sequence of events, the generator is called with this event after the *QrnDgEvent_04_End* event, or after an exception occurs.

Note: Callbacks cannot be called during this event.

Tracing a DATA-GEN operation

If you want to know the sequence of the calls from DATA-GEN to the generator and the calls from the generator back to DATA-GEN using the callback procedures, use the QIBM_RPG_DATA_GEN_TRACE environment variable to enable tracing. See the DATA-GEN section of the IBM Rational Development Studio for i: ILE RPG Reference.

Here is an example of a trace:

1. The generator has written its own trace message. The *nested* parameter was '0', so the trace message was printed started in column 1.
2. The generator has called the `QrnDgAddText` callback procedure.
3. The generator has written its own trace message. The *nested* parameter was '1', so the trace message was printed at the current indent level.

```
Start DATA-GEN
 Event 3 (Start)
  Event 5 (StartStruct) for ds
Terminate event enabled                1
   Event 11 (ScalarValue) for item
    AddText: 'Book'                    2
   Event 11 (ScalarValue) for price
    AddText: '25.99'                   2
   Event 11 (ScalarValue) for discount
    Looking up the discount            3
    AddText: 'sub=-12.50'              2
  Event 6 (EndStruct) for ds
 Event 4 (End)
 Event 12 (Terminate)
End DATA-GEN
```

Note: The trace output may not show up immediately, or it may flash by too quickly to see. For information on how to handle this situation, see "Tracing a DATA-INTO parser" on page 33.

Example of a DATA-GEN generator

Note: Detailed explanation is provided only for the aspects of the example that are related to the DATA-GEN operation.

In this example, a generator generates an HTML table for the DATA-GEN operation.

If you want to try running the code in the example, see "SQL statements to create the file used by the example" on page 61 for the SQL statements to create the `file` used by the program.

See "Output generated by the DATA-GEN operations in the example" on page 61 for the HTML generated by the program.

RPG program with DATA-GEN operations

The following shows the RPG program that uses the DATA-GEN operation.

Note the following aspects of the program:

1. The data structures are defined with a subfield for each row expected in the HTML table.

   The externally-described data structure uses EXTFLD statements to set the case of the names required for the column headings in the HTML table. For the *ITEMPRICE* field, the EXTFLD statement also adds an underscore. The generator replaces underscores with blanks when generating the column headings.
2. For the first three DATA-GEN operations, the output file is specified in the first operand of the %DATA built-in function. Option "doc=file" indicates that the first operand is the name of a file.
3. The program that does the generation is specified as the first operand of the %GEN built-in function. See "Program to generate an HTML table" on page 52 for the source for the program. The program that does the generation supports an optional character or UCS-2 value as the second operand of the %GEN built-in function. This value is used as the caption for the table.
4. The DATA-GEN *START operation starts a DATA-GEN sequence.

5. The next DATA-GEN operation continues the sequence. It writes out a row in the HTML table.
6. The DATA-GEN *END operation ends the sequence.
7. The final DATA-GEN is not part of a sequence. The result of the DATA-GEN operation is put in the variable specified as the first operand of the %DATA built-in function.

```
**free

DCL-C FILENAME 'MYORDERS';
DCL-F orders EXTDESC(FILENAME)
             EXTFILE(*EXTDESC);
DCL-DS order EXTNAME(FILENAME : *INPUT)             // 1
             QUALIFIED;
   Name extfld('NAME');
   Type extfld('ITEMTYPE');
   Item_Price extfld('ITEMPRICE');
END-DS;
DCL-DS customer QUALIFIED;                          // 1
   Name VARCHAR(30);
   Address VARCHAR(100);
   Zip_Code PACKED(9);
END-DS;

DCL-S customerTable VARCHAR(1000);

DATA-GEN *START %DATA('order.html' : 'doc=file')    // 2, 4
                %GEN('GENHTMLTAB'
                   : 'Order for ' + %CHAR(%DATE()));// 3

READ orders order;
DOW NOT %EOF;
   DATA-GEN order %DATA('order.html'
                      : 'doc=file output=continue') // 2, 5
                  %GEN('GENHTMLTAB');               // 3
   READ orders order;
ENDDO;

DATA-GEN *END %DATA('order.html' : 'doc=file')      // 2, 6
              %GEN('GENHTMLTAB');                   // 3

customer.Name = 'A. Smith';
customer.Address = '123 Elm Street';
customer.Zip_Code = 11111;
DATA-GEN customer %DATA(customerTable)              // 7
                  %GEN('GENHTMLTAB');               // 3

*INLR = '1';
```

Program to generate an HTML table

Note the following aspects of the initial section of the module:

1. This generator is a program.
2. Since the *name* and *value* subfields of the parameter passed to the generator have UTF-16 data, it is convenient to set the default CCSID for UCS-2 items to UTF-16.
3. Copy member QRNDTAGEN in file QOAR/QRPGLESRC defines the parameter passed to the generator and named constants for other information needed by the generator.
4. The *state_t* data structure define information used by the generator to keep track of the generation.
5. Several error codes and matching messages are defined for the errors detected by this generator.

```
**free

CTL-OPT OPTION(*SRCSTMT);
CTL-OPT MAIN(genHtmlTab);            // 1
CTL-OPT CCSID(*UCS2 : *UTF16);       // 2
/IF DEFINED(*CRTBNDRPG)
   CTL-OPT DFTACTGRP(*NO);
/ENDIF

/copy QOAR/QRPGLESRC,QRNDTAGEN       // 3

DCL-C MAX_CAPTION 10000;

DCL-DS state_t qualified template;   // 4
   haveHeader IND;
   inDataStructure IND;
   dataStructureName LIKE(QrnDgName_t);
   numSubfields INT(10);
   caption VARUCS2(MAX_CAPTION);
   haveCaption IND;
END-DS;

DCL-DS errorCodes qualified;         // 5
   DCL-DS nestedStructNotAllowed;
      code INT(10) INZ(1);
      msg VARCHAR(100) INZ('Nested structures are not allowed.');
   END-DS;
   DCL-DS differentStruct;
      code INT(10) INZ(2);
      msg VARCHAR(100)
          INZ('The data structure is not the same.');
   END-DS;
   DCL-DS eventNotSupported;
      code INT(10) INZ(3);
      msg VARCHAR(100) INZ('The event is not supported.');
   END-DS;
   DCL-DS valueNotInStruct;
      code INT(10) INZ(4);
      msg VARCHAR(100) INZ('All values must be subfields.');
   END-DS;
   DCL-DS userParmTypeNotSupported;
      code INT(10) INZ(4);
      msg VARCHAR(100) INZ('The user-parm must be UTF-16 or alphanumeric with the job
CCSID.');
   END-DS;
END-DS;
```

Main procedure

Note the following aspects of the procedure:

1. A single parameter is passed to the generator.
2. The *state* data structure holds state information maintained by the generator for all calls to the generator for the DATA-GEN operation or the sequence of DATA-GEN operations.

   The data structure is based on a pointer, since the storage for the data structure will be allocated from the heap.
3. Setting pointer *pEnv* from the *env* subfield of the parameter passed to the generator allows the generator to call the callback procedures

   Warning: The *env* pointer is null for the QrnDgEvent_12_Terminate event.

   Do not attempt to call any callback procedures during this event.
4. The generator needs to deallocate the state pointer when the generation is complete, so it enables the QrnDgEvent_12_Terminate event.
5. Callbacks are not allowed during the QrnDgEvent_12_Terminate event, so the generator just deallocates the state pointer and returns.

6. If the state information has not been allocated yet, the generator allocates and initializes the *generatorParm* pointer in the parameter passed to the generator.
7. The generator sets the basing pointer for the state information from the *generatorParm* pointer.
8. This event signals the beginning of a sequence of DATA-GEN operations.

   If the second operand of %GEN was specified, it is passed in the *userParm* subfield of the parameter passed to the generator. The generator will call the `getCaption` procedure to get the value of *userParm*.
9. This event signals the end of a sequence of DATA-GEN operations. The generator generates the end of the HTML table.
10. This event signals the beginning of the events for a DATA-GEN operation related to a variable.

    If the second operand of %GEN was specified, it is passed in the *userParm* subfield of the parameter passed to the generator. If the caption has not already been determined by a previous call to the generator, the generator will call the `getCaption` procedure to get the value of *userParm*.
11. This event signals the end of the events for a DATA-GEN operation related to a variable. If the DATA-GEN operation is not part of a sequence of DATA-GEN operations, the generator generates the end of the HTML table.
12. These events signal the beginning of an array of data structures or a single data structure. Since this might be the first time the generator has seen anything related to a data structure, it calls procedure *genStartTable* to generate the header for the HTML table, if necessary.

    If the *genStartTable* procedure detects an error in the calls to the generator, it will report the error to DATA-GEN, and control will not return to the statement following the call to the *genStartTable* procedure.
13. This event signals the end of an array of data structures. This generator does not need to do anything during this event.
14. The generator generates the beginning of a row in the HTML table.
15. This event signals the end of a data structure. This generator generates the end of the row in the HTML table.
16. This event signals a scalar value. If the generator detects that it is not a subfield, it issues an error.
17. The generator generates the beginning of the column in the HTML table.
18. The generator calls the `QrnDgAddText` callback procedure to generate the value for the column. The *value* parameter of the parameter passed to the generator points to UTF-16 data and the *valueLenChars* subfield has the number of double-byte characters in the UTF-16 data. These parameters can be passed directly to the `QrnDgAddText` procedure, since it expects a pointer to UTF-16 data and the number of double-byte characters.
19. The generator generates the end of the column in the HTML table.
20. The generator does not support any other events.

```
DCL-PROC genHtmlTab;
   DCL-PI *N;
      parm LIKEDS(QrnDgParm_t);                        //  1
   END-PI;

   DCL-DS state LIKEDS(state_t) based(pState);         //  2

   pQrnDgEnv = parm.env;                               //  3

   parm.doTerminateEvent = *ON;                        //  4

   IF parm.event = QrnDgEvent_12_Terminate;
      DEALLOC(N) parm.generatorState;                  //  5
      RETURN;
   ENDIF;

   IF parm.generatorState = *NULL;                     //  6
      parm.generatorState = %ALLOC(%SIZE(state_t));
      pState = parm.generatorState;
      CLEAR state;
   ENDIF;
   pState = parm.generatorState;                       //  7

   IF parm.event = QrnDgEvent_01_StartMultiple;        //  8
      IF parm.userParm <> *NULL;
         state.caption = getCaption (parm : state);
         state.haveCaption = *ON;
      ENDIF;

   ELSEIF parm.event = QrnDgEvent_02_EndMultiple;      //  9
      genEndTable (parm : state);

   ELSEIF parm.event = QrnDgEvent_03_Start;            //  10
      IF parm.userParm <> *NULL
      AND state.haveCaption = *OFF;
         state.caption = getCaption (parm : state);
         state.haveCaption = *ON;
      ENDIF;

   ELSEIF parm.event = QrnDgEvent_04_End;              //  11
      IF NOT parm.isPartOfSequence;
         genEndTable (parm : state);
      ENDIF;

   ELSEIF parm.event = QrnDgEvent_09_StartStructArray; //  12
      IF parm.userParm <> *NULL
      AND state.haveCaption = *OFF;
         state.caption = getCaption (parm : state);
         state.haveCaption = *ON;
      ENDIF;

      genStartTable (parm : state : parm.array.numSubfields);
      // Control may not return here

   ELSEIF parm.event = QrnDgEvent_10_EndStructArray;  // 13

   ELSEIF parm.event = QrnDgEvent_05_StartStruct;      //  12
      IF parm.userParm <> *NULL
      AND state.haveCaption = *OFF;
         state.caption = getCaption (parm : state);
         state.haveCaption = *ON;
      ENDIF;

      genStartTable (parm : state : parm.ds.numSubfields);
      // Control might not return here

      state.inDataStructure = *ON;                     //  14
      writeLine (parm : '<tr>');

   ELSEIF parm.event = QrnDgEvent_06_EndStruct;        //  15
      writeLine (parm : '</tr>');
      state.inDataStructure = *OFF;
   ELSEIF parm.event = QrnDgEvent_11_ScalarValue;      //  16
      IF NOT state.inDataStructure;
         error (parm : state
              : errorCodes.valueNotInStruct.code
              : errorCodes.valueNotInStruct.msg);
         // Control will not return here
      ENDIF;

      writeLine (parm : '<td>' : *ON);           //  17

      // The text for the column is written out in the same
      // UTF-16 CCSID as it was passed to the generator
      QrnDgAddText (parm.handle                        //  18
                  : parm.scalar.value
                  : parm.scalar.valueLenChars);
      writeLine (parm : '</td>');                //  19

   ELSE;
      error (parm : state                              //  20
           : errorCodes.eventNotSupported.code
           : errorCodes.eventNotSupported.msg);
      // Control will not return here
   ENDIF;
END-PROC genHtmlTab;
```

getCaption procedure

This procedure gets the value for the table's caption from the *userParm* subfield of the parameter passed to the generator. This subfield is set from the second parameter of the %GEN built-in function for the DATA-GEN operation.

Note the following aspects of the procedure:

1. The parameter passed to the generator has all the information a generator needs to interpret the *userParm* subfield if the second operand of %GEN was a string type. See "Types of the user-parameter passed to the DATA-GEN generator" on page 46.

   However, this generator does not support all possible types. It does not support alphanumeric values if the CCSID is not the job CCSID. It does not support UCS-2 values if the CCSID is not UTF-16.

   The *userParm* data structure is based on the *parm.userParm* pointer. A subfield is defined at position 1 of the data structure for every type of string that this generator supports.
2. The %STR built-in function returns the value of a null-terminated string.
3. If the alphanumeric value is in the job CCSID, the generator determines the length from the *parm.userParmSize* subfield, and uses that length to control the amount of data returned by the %SUBST built-in function.
4. If the alphanumeric value is in the job CCSID, the generator does not need to use the *parm.userParmSize* information, because the varying-length field holds its own length in its varying-length prefix.
5. For a UCS-2 or UTF-16 value, the *parm.userParmSize* must be divided by 2 to determine the number of characters for the %SUBST built-in function.
6. If the type or CCSID of the user-parameter is not supported by the generator, the generator issues an error.

```
DCL-PROC getCaption;
   DCL-PI *N VARUCS2(10000) EXTPROC(*DCLCASE);
      parm LIKEDS(QrnDgParm_t);
      state LIKEDS(state_t);
   END-PI;

   DCL-DS userParm QUALIFIED BASED(P);         // 1
      charFixed CHAR(MAX_CAPTION) POS(1);
      charVar2 VARCHAR(MAX_CAPTION) POS(1);
      charVar4 VARCHAR(MAX_CAPTION:4) POS(1);
      utf16Fixed UCS2(MAX_CAPTION) POS(1);
      utf16Var2 VARUCS2(MAX_CAPTION) POS(1);
      utf16Var4 VARUCS2(MAX_CAPTION:4) POS(1);
   END-DS;
   DCL-S CAPTION VARUCS2(MAX_CAPTION);
   DCL-S LEN INT(10);

   p = parm.userParm;
   IF parm.userParmType = QrnUserParmType_nullTerminatedString; // 2
      caption = %STR(parm.userParm);
   ELSEIF parm.userParmType = QrnUserParmType_char              // 3
   AND parm.userParmCcsid = QrnDg_JOB_CCSID;
      len = parm.userParmSize;
      caption = %TRIM(%SUBST(userParm.charFixed : 1 : len));
   ELSEIF parm.userParmType = QrnUserParmType_varchar_2         // 4
   AND parm.userParmCcsid = QrnDg_JOB_CCSID;
      caption = userParm.charVar2;
   ELSEIF parm.userParmType = QrnUserParmType_varchar_4
   AND parm.userParmCcsid = QrnDg_JOB_CCSID;
      caption = userParm.charVar4;
   ELSEIF parm.userParmType = QrnUserParmType_ucs2
   AND parm.userParmCcsid = 1200;
      len = parm.userParmSize / 2;                              // 5
      caption = %TRIM(%SUBST(userParm.utf16Fixed : 1 : len));
   ELSEIF parm.userParmType = QrnUserParmType_varucs2_2
   AND parm.userParmCcsid = 1200;
      caption = userParm.utf16var2;
   ELSEIF parm.userParmType = QrnUserParmType_varucs2_4
   AND parm.userParmCcsid = 1200;
      caption = userParm.utf16var4;
   ELSE;
      error (parm : state
           : errorCodes.userParmTypeNotSupported.code
           : errorCodes.userParmTypeNotSupported.msg);
      // Control will not return here
   ENDIF;
   RETURN caption;

END-PROC getCaption;
```

`genStartTable` procedure

This procedure generates the header for the HTML table, using the subfield names of the RPG data structure.

Note the following aspects of the procedure:

1. If the generator is already processing a data structure, the generator raises an error condition.
2. If the generator has already generated the header, and the name or the number of subfields of the current data structure is different from the data structure used to generate the header, the generator raises an error condition.
3. The generator sets the name and number of subfields of the current data structure in its state information, so it can ensure that the data structures it encounters are all the same.
4. The generator generates the beginning of the HTML table.
5. The generator generates a row in the header of the HTML table for each subfield of the data structure.
6. The generator calls the `QrnDgGetSubfieldName` callback procedure to obtain the name of the subfield.

7. The generator generates the end of the header for the HTML table.

```
DCL-PROC genStartTable;
   DCL-PI *n extproc(*dclcase);
      parm LIKEDS(QrnDgParm_t);
      state LIKEDS(state_t);
      numSubfields int(10) value;
   END-PI;
   DCL-S i INT(10);

   IF state.inDataStructure;                        // 1
      error (parm : state
           : errorCodes.nestedStructNotAllowed.code
           : errorCodes.nestedStructNotAllowed.msg);
      // Control will not return here
   ENDIF;

   if state.haveHeader;
      if parm.name <> state.dataStructureName      // 2
      or numSubfields <> state.numSubfields;
         error (parm : state
              : errorCodes.nestedStructNotAllowed.code
              : errorCodes.nestedStructNotAllowed.msg);
      endif;

      return;
   endif;

   state.dataStructureName = parm.name;            // 3
   state.numSubfields = numSubfields;
   state.haveHeader = *ON;

   writeLine (parm                                 // 4
            : '<table border="1">');

   if state.haveCaption;
      writeLine (parm : '<caption>' : *on);
      QrnDgAddText (parm.handle
                  : %addr(state.caption : *data)
                  : %len(state.caption));
      writeLine (parm : '</caption>');
   endif;

   writeLine (parm
            : '<thead>'
            + '<tr>');

   FOR i = 1 TO numSubfields;                      // 5
      writeLine (parm
              : '<td><b>'
              : *ON); // skip the newline for this output
      genColumnName (parm
                   : state
                   : QrnDgGetSubfieldName (parm.handle : i));  // 6
      writeLine (parm : '</b></td>');
   ENDFOR;

   writeLine (parm                                 // 7
            : '</tr>'
            + '</thead>'
            + '<tbody>');

END-PROC genStartTable;
```

genColumnName procedure

This procedure generates the name of a column from the name of a subfield.

Note the following aspects of the procedure:

1. This procedure expects a subfield to have its words separated by underscores, with the words capitalized as required for the table; for example *Column_Heading*. To create the column name, this procedure changes the underscores to spaces.

See the code for the program using this DATA-GEN generator to see how it defines the subfield names.

2. The *columnName* variable is a varying length UTF-16 variable, so the generator can use the `QrnDgAddText` callback procedure to generate the name for the column.

```
DCL-PROC genColumnName;
   DCL-PI *n extproc(*dclcase);
      parm LIKEDS(QrnDgParm_t);
      state LIKEDS(state_t);
      name like(QrnDgName_T) const;
   END-PI;
   DCL-S columnName LIKE(name);
   DCL-C underscore %UCS2('_');
   DCL-C BLANK %UCS2(' ');

   columnName = %XLATE(UNDERSCORE : BLANK : name);       // 1

   QrnDgAddText (parm.handle                             // 2
               : %ADDR(columnName : *DATA)
               : %LEN(columnName));
END-PROC genColumnName;
```

`genEndTable` procedure

This procedure generates the end of the HTML table.

```
DCL-PROC genEndTable;
   DCL-PI *n extproc(*dclcase);
      parm LIKEDS(QrnDgParm_t);
      state LIKEDS(state_t);
   END-PI;

   writeLine (parm
            : '</tbody>'
            + '</table>');
END-PROC genEndTable;
```

`writeLine` procedure

This procedure generates text in the job CCSID.

Note the following aspects of the procedure:

1. The generator determines whether it should generate a new-line after it generates the text. It first uses the *outputIsToFile* subfield of the parameter passed to the generator to determine whether the generated text is intended for a stream file. If not, then a new-line is not needed.

   However, the caller may have indicated that this procedure should not generate a new-line in the optional *skipNewLine* parameter.
2. Since the text is in the job CCSID, the generator can use the simple `QrnDgAddTextString` callback procedure to output the text.
3. The generator adds the new-line character using the `QrnDgAddTextsNewLine` callback procedure.

```
DCL-PROC writeLine;
   DCL-PI *N EXTPROC(*DCLCASE);
      parm LIKEDS(QrnDgParm_t);
      line pointer VALUE OPTIONS(*STRING);
      skipNewLine IND CONST OPTIONS(*NOPASS);
   END-PI;
   DCL-S doNewLine IND INZ(*ON);

   doNewLine = parm.outputIsToFile;               //  1
   IF %PARMS() >= %PARMNUM(skipNewLine)
   AND skipNewLine;
      doNewLine = *OFF;
   ENDIF;

   QrnDgAddTextString (parm.handle : line);       //  2
   IF doNewLine;
      QrnDgAddTextNewline (parm.handle);          //  3
   ENDIF;
END-PROC writeLine;
```

`error` procedure

This procedure raises an error condition which will cause the DATA-GEN operation to fail.

Note the following aspects of the procedure:

1. This procedure first outputs a trace message using the `QrnDgTrace` callback procedure. If the user is tracing the DATA-GEN operation, the trace message will explain the error code that will appear next in the trace.
2. The `QrnDgReportError` callback procedure is used to report the error condition. **ON* is passed for the *nested* parameter, indicating that the trace message should be nested within other information in the trace.

   Control will not return to this procedure after the call to the `QrnDgReportError` procedure. The generator will get called again for the final QrnDgEvent_12_Terminate event.

```
DCL-PROC error;
   DCL-PI *n extproc(*dclcase);
      parm LIKEDS(QrnDgParm_t);
      state LIKEDS(state_t);
      errorCode INT(10) VALUE;
      errorMessage VARCHAR(100) CONST;
   END-PI;

   QrnDgTrace (parm.handle                        //  1
             : errorMessage
             : '1');

   QrnDgReportError (parm.handle                  //  2
                   : errorCode);
   // Control will not return here

END-PROC error;
```

SQL statements to create the file used by the example

```
CREATE TABLE QGPL/MYORDERS
   (NAME VARCHAR (25 ) NOT NULL WITH DEFAULT,
   ITEMTYPE VARCHAR (25 ) NOT NULL WITH DEFAULT,
   ITEMPRICE DECIMAL (9 , 2) NOT NULL WITH DEFAULT)

INSERT INTO QGPL/MYORDERS
    VALUES('Refrigerator', 'Appliance', 525.95)

INSERT INTO QGPL/MYORDERS
    VALUES('Shirt', 'Clothing', 5.95)

INSERT INTO QGPL/MYORDERS
    VALUES('Rake', 'Gardening', 15.95)
```

Output generated by the DATA-GEN operations in the example

The following is the HTML table generated by the DATA-GEN sequence.

```
<table border="1">
<caption>Order 2019-11-15</caption>
<thead><tr>
<td><b>Name</b></td>
<td><b>Type</b></td>
<td><b>Item Price</b></td>
</tr></thead><tbody>
<tr>
<td>Refrigerator</td>
<td>Appliance</td>
<td>525.95</td>
</tr>
<tr>
<td>Shirt</td>
<td>Clothing</td>
<td>5.95</td>
<td>5.95</td>
</tr>
<tr>
<td>Rake</td>
<td>Gardening</td>
<td>15.95</td>
</tr>
</tbody></table>
```

The following shows how the table appears:

Table 8. x

Name	**Type**	**Item Price**
Refrigerator	Appliance	525.95
Shirt	Clothing	5.95
Rake	Gardening	15.95

The following shows the value of the *customerTable* variable after the final DATA-GEN operation in the program, as shown in the debugger.

```
> EVAL customerTable
  CUSTOMERTABLE =
            ....5...10...15...20...25...30...35...40...45...50...55...60
       1   '<table border="1"><thead><tr><td><b>Name</b></td><td><b>Addr'
      61   'ess</b></td><td><b>Zip Code</b></td></tr></thead><tbody><tr>'
     121   '<td>A. Smith</td><td>123 Elm Street</td><td>11111</td></tr><'
     181   '/tbody></table>                                             '
```

The following shows how the table appears:

Table 9. x

Name	**Address**	**Zip Code**
A. Smith	123 Elm Street	11111

Notices

Programming interface information

This Rational Open Access: RPG Edition publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of Rational Open Access: RPG Edition.

Trademarks

Terms and conditions

RZAS-M000-02