System i Developer

What's New in RPG?
Data-Into, Free-format & More

Jon Paris

Jon.Paris @ Partner400.com
www.Partner400.com
www.SystemiDeveloper.com

Partner400

Notes

Partner400

In recent years RPG has continued to add new features including the latest enhancement DATA-INTO.

In this session Jon will explain how this new feature works and how you can use it to process everything from CSVs to JSON. Topics to be covered include:

- Basic syntax and usage rules
- Details of the supplied parsers
- How to write your own parser to extend the functionality

Of course DATA-INTO is not the only new feature in RPG, so we'll take a few minutes to quickly run through some other recent enhancements. Topics to be covered include:

- Fully free-form RPG - with no column limitations
- The new ON-EXIT support for application clean-up
- Enhancements in null-indicator support
- Full support for longer SQL column names via ALIAS support
- And more if we have time ...

The authors, Jon Paris and Susan Gantner, are co-founders of Partner400, a firm specializing in customized education and mentoring services for IBM i developers. Together with Paul Tuohy, they also operate the System i Developer education consortium which runs the RPG & DB2 Summit conferences. See systemideveloper.com/Summit/conferences.html.

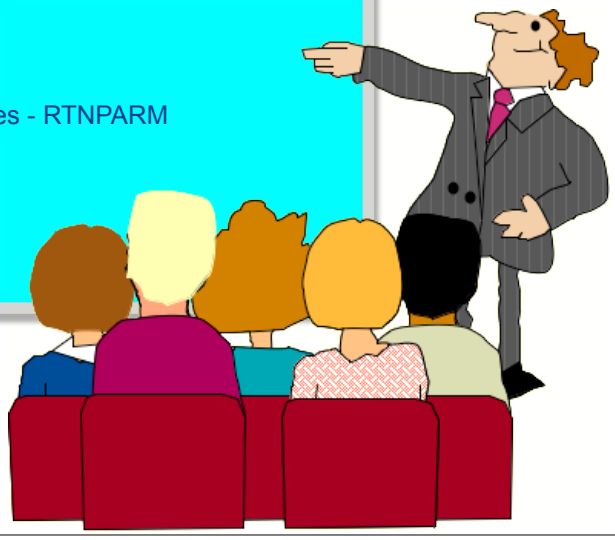
Their individual careers include a number of years with IBM, including working at both the Rochester and Toronto laboratories. These days Jon and Susan are devoted to educating developers on techniques and technologies to extend and modernize their applications and development environments.

Jon and Susan author regular technical articles for the IBM publication, IBM Systems Magazine and the companion electronic newsletter Extra. You may view their past articles at ibmsystemsmag.com/authors/susan-gantner/ or ibmsystemsmag.com/authors/jon-paris/ They also write a regular blog which you can find at: ibmsystemsmag.com/blogs/idevelop/

This presentation may contain small code examples that are furnished as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. We therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Most Recent RPG Enhancements

- DATA-INTO
- ON-EXIT
- %PROC BIF
- %Max, %Min BIFs
- Data Definition and Usage Enhancements
  - Sort and Search Data Structure Arrays
  - Support For Alias Names
- New %SCANR BIF
  - And enhancement to %SCAN
- Performance boost for large return values - RTNPARM
- Better NULL support
- Fully free-form RPG:
  - Complete column freedom
  - Including true Nested Data Structures



Notes

We don't have time to go into all of these features in depth but we have written articles on many of them and of course you can find more about any that I skip by checking out the V7 RPG manual in the Information Center.

DATA-INTO - Handling "Other" Data

The DATA-INTO operation is similar to the XML-INTO operation

- It "unpacks" data from a document into a data structure

It is different from XML-INTO in the following important ways:

- It allows processing of any type of hierarchical information
  - JSON documents, Properties files, CSV (Character-separated-values)
  - ... you name it
- IBM does not provide the parser(s)
  - You supply them - by writing one or obtaining one elsewhere
- But IBM has provided sample parsers for illustrative purposes
  - Including a JSON parser and a simple Properties parser

DATA-INTO is available on 7.2 and 7.3

- Appropriate PTFs are required
- Details at ibm.biz/data-into-rpg-opcode-ptfs

Notes

Rather than create another specific data format parser, such as JSON-Into or CSV-Into, etc, IBM decided to follow the path that they had set with Open Access and design this new capability as a language extension point. This not only allows companies to write their own parsers, but also allows third parties and open-source groups to offer parsers for a variety of data interchange formats. To get the ball rolling, IBM is supplying a number of sample parsers. While many are primarily intended as teaching tools, they are also supplying a complete JSON parser, which should make those who lobbied for JSON-INTO happy.

A link to some details from IBM on using DATA-INTO, including the necessary PTFs to have installed for it to work can be found here: ibm.biz/data-into-rpg-opcode-ptfs

DATA-INTO - XML-INTO Meets Open Access

Like XML-INTO, it parses data into an RPG DS or array

Like Open Access, logic to make it work is not supplied by the RPG compiler

- It is user-written (or obtained elsewhere)
- So it works for any format of data
  - CSV, JSON, your own proprietary style of data interchange, etc.

Sample1.csv

```
1234,Jones
2345,Smith
3456,Gantner
4567,Paris
```

```
dcl-ds accounts Qualified  Dim(10)  Inz;
    account      char(4);
    name         char(20);
end-ds;

data-into accounts
  data('Sample1.csv': 'doc=file case=any ccsid=job')
  %parser('*LIBL/PARSECSV1');
```

Contents of Accounts DS after Data-Into

```
accounts(1).account = '1234'
accounts(1).name = 'Jones'
accounts(2).account = '2345'
accounts(2).name = 'Smith'
accounts(3).account = '3456'
accounts(3).name = 'Gantner'
accounts(4).account = '4567'
accounts(4).name = 'Paris'
```

The logic to parse the csv data is here

Notes

The best way to think of DATA-INTO is as a combination of Open Access and XML-INTO. XML-INTO takes data and unpacks it into a matching RPG data structure. Open Access utilizes a custom handler, written by you or supplied by a third party, to treat data originating from any source as if it came from a file. DATA-INTO places data into Data Structure (or array), as XML-INTO does but it uses a custom parser to figure out what data goes where, as Open Access does.

Our first simple example here is a bit different from many kinds of data (such as JSON or properties) that you may be handling in that there are no names associated with each "field" of data. It consists of a four-character account code followed by a name of arbitrary length. The important thing to note here is that, unlike an XML or JSON document, there is no name associated with the individual pieces of data. Their position in the record determines which field they represent.

DATA-INTO Syntax

Very similar to XML-Into

- %Data replaces %XML
- And %Parser is added to identify the program/procedure to perform the parsing
- Many of the options keywords are the same as XML-Into as well
- Just as with XML-INTO there is also a %HANDLER parameter variant

Basic Syntax:

```
DATA-INTO{(EH)}  receiver
        %DATA( document  {:  options  })
        %PARSER( parser  {:  parser  options  }};
```

- Options include:
  - doc = file means the first parameter refers to an external file name
  - case=any means when names are used they are not case sensitive

```
DATA-INTO accounts
     %data('Sample1.csv': 'doc=file case=any')
     %parser('*LIBL/PARSECSV1');
```

Notes

In the example shown, the Data Structure (DS) accounts is being loaded with data extracted from the file sample1.csv. The work of identifying the fields within that file and their associated values is to be performed by the parser PARSECSV1. As you can see in the syntax diagram it is also possible for the caller to pass additional parameter information to the parser via the parser options parameter.

In addition to the basic syntax shown here, like XML-INTO before it, DATA-INTO also allows for a %HANDLER variant for those instances where the amount of data to be processed exceeds RPG's capacity limitations, or where you simply want to process each portion of the data as and when it is available.

DATA-INTO Parsers

IBM Supplies three sample parsers

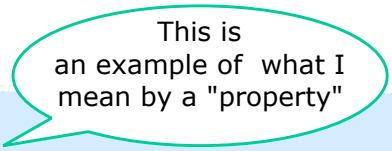
- A JSON parser
- Two property parsers
  - One that processes a string of properties with a character separator
  - And one that has one property per record

Scott Klement has also released a DATA-INTO parser for JSON

- It is based on his port of the YAJL tool set

I have published two CSV parsers

- References to those articles appear later in the presentation
  - And we will study some of the code in one of them




```
DATA-INTO accounts
     %data('Sample1.csv': 'doc=file case=any')
     %parser('*LIBL/PARSECSV1');
```

Notes

IBM's parsers are supplied in source code form and are intended as examples of how parsers should be written - they are NOT intended for production use as-is. The JSON parser in particular has been written to simplify the target data structure and therefore requires some "tweaking" in oder to handle some fairly standard JSON formats.

Scott's YAJL port offers an option to allow it to work as simply as IBM's does but also handles all valid JSON documents.

My own parsers are intended as educational. They have however been adopted for production use by a number of companies with minimal modifications. At some point I intend to offer a "cleaned up" production ready version as an open source project.

Writing a Data-Into Parser (1 of 3)

RPG extracts the data from the variable or file specified

- Loads it in a buffer which is passed to the parser

The parser's job is to process the buffer

- Notifying RPG about what it finds there

It does this by calling a set of procedures

- All parsers will need to use the following procedures:
  - QrnDiStart() to report that processing has started
  - QrnDiFinish() says parsing is complete and to return control
- Other procedures are then based on the data content
  - They are discussed on the next chart ...

For more details on the Data-Into and the example shown here, see:
ibmsystemsmag.com/ibmi/developer/rpg/rpg-data-into/
ibmsystemsmag.com/ibmi/developer/rpg/closer-look-data-into/

Notes

When you invoke DATA-INTO, RPG runtime logic first extracts the data from the variable or the file that you specified and places it in a buffer. It then passes the address of that buffer, along with its length and other identifying information, to your parser.

Once in control, the parser processes the buffer notifying RPG what it has found. In order to do this, it can call a number of procedures which allow it to inform RPG about the structure of the data, names, and their values. We'll talk a little more about these procedures when we look at a simple example later. For the moment though, let's simply step through the major ones that just about every parser will use, in the sequence in which they will typically be used.

The parser must start by calling QrnDiStart() to notify RPG that processing has commenced.

At the end of the processing, QrnDiFinish() is called to notify RPG that processing is finished and it should return control back to the program that issued the Data-Into operation.

Writing a Data-Into Parser (2 of 3)

Partner400

After calling QrnDiStart()

- If parsing repeating elements, declare the start of an array
  - This includes Data Structure arrays
  - QrnDiStartArray() is used for this
    - QrnDiEndArray() identifies the end of the array
- If there is a structure, declare the start of a structure
  - If a DS array, declare the structure after declaring the array
  - QrnDiStartStruct() reports that we're starting a data structure
    - QrnDiEndStruct() declares the end of the structure
    - In the case of a DS array, this is the end of one element of the array

Note: most documents will likely be using DS Arrays

Reporting of the actual data for each element found during the parsing process is covered on the next chart

Notes

Partner400

If the data being parsed is an array, including a DS array, then QrnDiStartArray() would be called.

If the data being parsed is a Data Structure (DS) then the call is to QrnDiStartStruct().

In the case of a DS array as in our example, we call ...StartArray() first followed by ...StartStruct().

Array elements and/or data structure subfields will be processed one at a time, first supplying the name of the item and then the value to be placed in that named item in the original program. Details of procedures to do that are on the next chart.

Once all of the fields have been processed, the parser would then call QrnDiEndStruct() to notify RPG that the structure (or, in the case of a DS array, an array element) has been completed.

Last but not least, the parser notifies RPG that it has completed its work by calling QrnDiFinish(). This tells RPG to return control to the original program once the parser exits.

At this point your main program regains control at the instruction following the DATA-INTO operation and your data should all be tucked away nice and neatly in its associated variables just waiting for you to process it.

Writing a Data-Into Parser (3 of 3)

After calling QrnDiStartArray() and/or QrnDiStartStruct() ...

- We're ready to begin reporting actual data
- Most likely this will be a subfield of a DS or DS Array

We report values in pairs (name and then value)

- Just like XML-INTO the matching is based on names
  - And hierarchy
- QrnDiReportName() reports the name of the item (subfield)
- QrnDiReportValue() supplies the value of that item

Rinse and repeat

- ...ReportName() and ...ReportValue for each and every subfield

Notes

When reporting data back to the RPG program, we do it by specifying the name of the item followed by the value of that item. QrnDiReportName() identifies a field within a DS or array and would immediately be followed by a call to QrnDiReportValue() to notify RPG of the value to be placed in that field. This pair of calls would be repeated for all fields in the DS or array.

Once all of the fields have been processed, the parser would then call QrnDiEndStruct() to notify RPG that the structure (or, in the case of a DS array, an array element) has been completed, as shown on an earlier chart.

After all the elements in an array (or DS array) have been processed, call to QrnDiEndArray() is needed, as shown on an earlier chart.

A Reminder ... Our Example

In this case, the csv being processed has no names with the values

- e.g. No column headings

So either the parser must make up the field names

- e.g. Column1, Column2, etc.

Or the parser must be file specific

- And that is the way this example has been written

See the notes page for other options

Sample1.csv

```
1234,Jones
2345,Smith
3456,Gantner
4567,Paris
```

```
dcl-ds accounts Qualified  Dim(10)  Inz;
    account      char(4);
    name         char(20);
end-ds;

data-into accounts
  data('Sample1.csv': 'doc=file case=any ccsid=job')
  %parser('*LIBL/PARSECSV1');
```

Contents of Accounts DS after Data-Into

accounts(1).account = '1234'	accounts(1).name = 'Jones'
accounts(2).account = '2345'	accounts(2).name = 'Smith'
accounts(3).account = '3456'	accounts(3).name = 'Gantner'
accounts(4).account = '4567'	accounts(4).name = 'Paris'

Notes

We wanted to keep the example as simple as possible, so the names of the items being read from the CSV are going to be supplied by the parser simply based on the position of the value in the sequence. That allows us to see the basic mechanics in progress without getting bogged down in how the names are obtained.

As a result the parser is not at all generic but ... there are two ways in which we could easily make it so.

We could modify the code to accept the column headings from the first record. Store them and then use the names in sequence as each value is extracted from subsequent records.

Alternatively (or in addition) we could use the ability to pass in an additional parameter to the parser and include the column names (i.e. field names) via that parameter.

If we were to combine these two ideas we could produce a truly generic parser that could process any CSV with or without a column heading row.

A Simple Parser, Data Definitions

For each field we will be populating, we defined variables to:

- Contain the name of the fields and their values

In addition we also have variables for:

- The record extracted from the RPG supplied buffer
- The position of the field separator (a comma in our case)
- A pointer to the current position within the buffer

```
(A)   // Names of DS subfields
      dcl-s subfieldName_Account  varchar(15)  inz('account');
      dcl-s subfieldName_Name     varchar(15)  inz('name');

(B)   // Values of subfields
      dcl-s account  varchar(10);
      dcl-s name     varchar(30);

(C)   // Variables for record location and content
      dcl-s record     varchar(256);
      dcl-s separator  int(5);

      dcl-s pcurrentPosn  pointer;
```

Notes

At (A) we define the names of the DS subfields that we will be populating. Remember, as we noted earlier, there are no field names in the data, so the parser must supply them. In a future example, we'll show you how we could use column names to get round this problem but for now we'll keep things simple. Notice that the fields are defined as varying length (varchar). We did this because, later on, we'll need both the name and its length which can easily be obtained by using %Len. By using this approach, we avoid having to remember to trim trailing spaces from the names all the time. Failure to do so would result in a field name mismatch within the RPG runtime.

The storage for the values of the account and name field are defined at (B). These are also varying length fields. In addition to making it easy to obtain the lengths of the values, this has the added advantage that should one of the target fields in the calling program's DS array be varying in length it will be handled correctly and not have a lot of spurious blanks in the back end of the field.

The variables at (C) are used for the current record, the position of the comma separator, and the position within the buffer at which the first/next record starts.

A Simple Parser, Logic (Part 1 of 4)

Partner400

Initial setup

- (D) Enable access to the QrnDixxxxx functions
- (E) Set the current position pointer to the value given to us by RPG
- (F) Notify RPG that we're beginning the parsing operation
- (G) Identify RPG that we'll be processing an array (a DS array in our example)
- (H) Extract the first record from the buffer

```
     // Enable access to the QrnDixxxxx functions
(D)  pQrnDiEnv = parm.env;

     // Set current record position to start of buffer
(E)  pCurrentPosn = parm.data;

     // Start the parse
(F)  QrnDiStart (parm.handle);

     // Notify RPG that the data represents an array
(G)  QrnDiStartArray(parm.handle);

   // Get first set of data to parse
(H)  record = getRecord(pcurrentPosn);
```

Notes

Partner400

Now that we've seen the basic variable definitions it is time to look at the program logic. As you'll see it follows very closely along the pattern outlined in the previous section.

The very first step (D) is to copy the environment pointer so that our code has access to all of the QrnDi... procedures. Don't worry about this—you simply need it at the start of every parser. Similarly, at (E) we copy the pointer to the data buffer that RPG has supplied into our current record position pointer.

Now the real work begins with a call to the QrnDiStart procedure (F). The parameter passed here is a "handle" that RPG has given us to uniquely identify this specific DATA-INTO operation. As you will see we have to include this value on each and every subsequent call to any of the QrnDixxx procedures.

At (G) we call the QrnDiStartArray procedure to notify RPG that we are starting an array. Since there's no array name in the file, we are simply starting an unnamed array. RPG will derive the name to be used as the target from the DATA-INTO operation.

A Simple Parser, Logic (Part 2 of 4)

Loop through the data elements in the buffer

```
    DoW record <> '';

      // find position of comma in record and extract values
(I)   separator = %scan(',': record);

      account = %subst(record: 1: separator - 1);
      name = %subst(record: separator + 1);

(J)  // Notify RPG of the beginning of an iteration of the structure
      QrnDiStartStruct (parm.handle);


      << See next chart for details of reporting each subfield >>
                  << That logic goes here. >>


      // End this iteration of structure
(L)   QrnDiEndStruct (parm.handle);

      // Get another set of data to parse
      record = getRecord(pcurrentPosn);

    EndDo;
```

Notes

Within the loop (I) we search for the position of the comma separator, and split the record into its composite fields. Once we have the data separated it is time to tell RPG what we have "discovered."

For each array element, we need to call QrnDiStartStruct (J) to notify RPG of the beginning of an array element.

Next, we will call QrnDiReportName and QrnDiReportValue to notify RPG of the names and values for the data that we have extracted. Because this chart was getting crowded, we have moved this logic to the next chart and we'll discuss the details of it there.

Once the values have been set, we call QrnDiEndStruct (L) to notify RPG that this element of the DS has been completed. We then simply move on to the next record and repeat the process.

A Simple Parser, Logic (Part 3 of 4)

This logic goes in the middle of the loop on the previous chart

It notifies RPG of the name of each field and its associated data

```
// Report the name of the 1st subfield 'account'
QrnDiReportName (parm.handle
                 : %addr(subfieldName_Account: *data)
                 : %len(subfieldName_account) );

// and give RPG the associated value
QrnDiReportValue(parm.handle
                 : %addr(account: *data)
                 : %len(account) );


// Repeat the process with the subfield 'name'
QrnDiReportName (parm.handle
                 : %addr(subfieldName_Name: *data)
                 : %len(subfieldName_Name) );

QrnDiReportValue(parm.handle
                 : %addr(name: *data)
                 : %len(name) );
```

Notes

We're inside the loop shown on the previous chart. This is the logic omitted from that chart.

For each "field" of data we find, we call (K) QrnDiReportName and QrnDiReportValue to notify RPG of the names and values for the data that we have extracted. Note that because we are using variable length fields for the names and values, the address we need to pass back to RPG is to the beginning of the data portion of the field—hence the use of the qualifier *data on the %Addr function.

A Simple Parser, Logic (Part 4 of 4)

(M) Notify RPG that all the DS array elements have been processed

(N) Finally notify RPG that we are all done

At this point control returns to the point following the DATA-INTO operation

For the full details of this example read:

- ibmsystemsmag.com/ibmi/developer/rpg/rpg-data-into/

And for a more comprehensive example the follow-up article:

- ibmsystemsmag.com/ibmi/developer/rpg/closer-look-data-into/

Download links to the code associated with the articles are included

```
    // Notify RPG of end of array
(M) QrnDiEndArray(parm.handle);

    // Notify RPG that we are ending the parse (no more data)
(N) QrnDiFinish(parm.handle);
```

Notes

Finally, when all records have been processed and the Do loop exits, we call QrnDiEndArray (M) to finish the array.

The final step is to call QrnDiFinish (N) to notify RPG that parsing has been completed. We then simply return from the parser, RPG "tidies up" and returns control to the statement following the DATA-INTO operation.

V7.3 New ON-EXIT Support

A great addition to the language for all those "clean up" tasks

- For example when a subprocedure blows up with an error

Typical Clean-up Uses

- Ensure that files are closed
- Release any dynamic storage that you have allocated
- Delete partial IFS files
- Clean up spool files
- Ensure that error is logged

Any code in the ON-EXIT section will always be executed

- Whether the routine ends normally (i.e. with a RETURN operation)
- Or because of an error (e.g. an uncaught divide by zero)

Notes

Full details can be found in the latest RPG IV Reference manual or in the RPG Cafe at:
ibm.biz/RPG_ON_EXIT_Section

These are the PTFs required to implement this support:

Release 7.2:
SI62949: RPG runtime
SI62955: TGTRLS(*CURRENT) compiler
Or, get DB2 PTF group SF99702 Level 14. These PTFs are part of that group PTF.

Release 7.3:
SI62950: RPG runtime
SI62957: TGTRLS(*CURRENT) compiler
SI62965: TGTRLS(*PRV) compiler
Or, get DB2 PTF group SF99703 Level 3. These PTFs are part of that group PTF.

V7.3 New ON-EXIT Support

Partner400

An optional Indicator variable can be specified

- It will be set to true (*On) if the exit was due to an error
- If the exit was normal then it will be false (*Off)

Must be coded at the end of the procedure

- Cannot use in the main logic of a "Cycle Main" procedure
  - i.e. A conventional RPG program

A RETURN operation can be included if required

- This allows you to ensure that a return value is supplied
  - Perhaps to indicate the error as shown in the following example
- Or even to override the value currently set to be returned
  - Although I am not yet sure why I would want to do that

Notes

Partner400

The exit type indicator on the operation is useful because it enables you to determine if the procedure is returning normally or because of an error. By testing it you can, for example, ensure that an appropriate value is returned from the procedure.

This support helps to round out the error handling built into RPG IV - there really is no reason any longer for any program to simply die with the green-screen-of-death when it is so easy to monitor and control errors.

Error Handling With MONITOR

MONITOR can be used to trap errors

- But it can't trap all abnormal terminations

```
dcl-proc AveragePrice1;
  dcl-pi AveragePrice1  packed(5:2);
      itemPrice  packed(5:2) Dim(40);
      itemCount  int(5);
  End-Pi;

dcl-s  totalPrice  packed(9:2) Inz;
dcl-s  averagePrice  packed(5:2);
dcl-s  i           int(3) Inz;

For i = 1 to itemCount;
  totalPrice += itemPrice(i);
EndFor;

Monitor;
  averagePrice = totalPrice / itemCount;

On-Error;
  Return -1;  // Return "impossible" average to indicate error

EndMon;

  // No error triggered so return average price
Return averagePrice;
```

Notes

This example shows how a subprocedure can make use of the MONITOR operation to handle things like divide by zero. Since that is the primary intent of this particular error handling routine I would normally have coded it to monitor specifically for divide by zero but in this example just took the easy way out and had it trap for all errors.

But what if the addition of the prices to the totalPrice variable had exceeded it's capacity. Right now that can't happen because there are only 40 prices, and the total has a capacity 100 times larger than any individual price. But suppose that at some point in the future that somebody changes the procedure to accept 100 or 200 prices? Certainly it could occur then.

I've always been a fan of bullet proofing my code as much as possible, and this new support really helps in that regard. You'll see how it can be applied to this potential situation on the next chart.

An Alternative Approach Using ON-EXIT

This will achieve the same results and handle ALL errors

- Even those caused by external job termination

```
dcl-proc AveragePrice2;
  dcl-pi AveragePrice2  packed(5:2);
      itemPrice  packed(5:2) Dim(40);
      itemCount  int(5);
  End-Pi;

dcl-s  errorExit   ind;
dcl-s  totalPrice  packed(9:2) Inz;
dcl-s  averagePrice  packed(5:2);
dcl-s  i           int(3) Inz;

For i = 1 to itemCount;
  totalPrice += itemPrice(i);
EndFor;

averagePrice = totalPrice / itemCount;

return averagePrice;

on-exit errorExit;
  If errorExit;
  //      Problem reporting logic
    return -1';   // Return "impossible" average to indicate error
```

Notes

Not only does this example handle the problem with the totalPrice variable overflowing, it will also capture any and all errors that occur during the running of the procedure no matter what the cause.

I'm not suggesting that ON-ERROR replaces the need for MONITOR - but it is a very useful addition as a "catch all" defence mechanism.

New built-in function %PROC

%PROC returns the external name of the current procedure

```
 // No EXTPROC - maybe even no PR or PI
Dcl-Proc p1;
x = %proc();   // x = 'P1'

// PR has EXTPROC
Dcl-Pr p2 ExtProc('proc2');
   ...
Dcl-Proc p2;
x = %proc();  // x = 'proc2'

// PI has EXTPROC
Dcl-Proc p3;
Dcl-pi *n ExtProc('P3a');
x = %proc();  // x = 'P3a'
```

Notes

Retrieving the name of the current procedure was tricky to do before. Now we can simply use the built-in %Proc.

For a cycle-main procedure (that is the main procedure in a "traditional" RPG program without "NoMain" or "Main" specified in the Ctl-Opt or H spec, the external name of the procedure is the name of the module when it was compiled.

For a linear-main procedure the name supplied is the name of the procedure defined with the Main keyword in uppercase.

For a subprocedure where EXTPROC was not specified, the external name of the procedure is the uppercase form of the name of the procedure.

For a subprocedure where EXTPROC was specified, the external name of the procedure is the value specified by EXTPROC.

Two New BIFs

%MAX and %MIN

- Return the value of the highest / lowest item in a list

Can also be used in declarations!

- e.g. to set the DIM of an array to the size of the largest in a list of arrays

```
Dcl-S  Q1_Sales  Packed(9:2);
Dcl-S  Q2_Sales  Packed(9:2);
Dcl-S  Q3_Sales  Packed(9:2);
Dcl-S  Q4_Sales  Packed(9:2);

Dcl-S  bestQuarter   Packed(9:2);
Dcl-S  worstQuarter  Packed(9:2);

Dcl-S  array1  Int(4)  Dim(20);
Dcl-S  array2  Int(8)  Dim(50);

Dcl-S  resultArray Int(8)  Dim( %Max( %Elem(array1): %Elem(array2));

bestQuarter  = %Max( Q1_Sales: Q2_Sales: Q3_Sales: Q4_Sales );

worstQuarter = %Min( Q1_Sales: Q2_Sales: Q3_Sales: Q4_Sales );
```

Notes

These are among the latest RPG enhancements. They became available on March 30th via the PTFs listed here:
http://ibm.biz/spring_2017_rpg_enhancements

This release also includes the new nested data structure capabilities that I mentioned earlier.

Sorting Data Structure Arrays

Partner400

Data structure arrays can now be sorted

- Using any of the subfields as the key
  - But only a single subfield can be used
    - No direct support for sorting multi-dimensional arrays
- Asterisk (*) identifies the level at which the array is to be sorted

SORTA can now have sequence specified

- Using the Op-code extenders A(scending) or D(escending)
- Can only be used when no sequence specified in the D-specs

```
Dcl-DS  ProductInfo2  Qualified  Dim(1000);
  Name         Char(8);
  UnitPrice    Packed(7:2);
  QtyInStock   Packed(9);
End-DS;

SortA ProductInfo2(*).UnitPrice;

SortA ProductInfo2(*).Name;
```

Notes

Partner400

With the advent of V5R2, it became possible to define Data Structure arrays. i.e. with a DIM keyword at the DS level. But at the time IBM did not provide any means by which such arrays could effectively be sorted. To do that you had to resort to using the qsort function. That shortcoming is removed in V7 and you can now sort on any subfield in the array. For example, given the DS array here, you can perform a sort on qtyInStock or totalSales or any of the other fields in the DS. As you can see from this code, the level of the array to be sorted is indicated by an asterisk (*) in the subscript position.

In the first example the DS array is sequenced on the totalSales values, and in the second the description. Another nice addition to the SORTA repertoire is that you can now specify whether the sort is to be in ascending or descending sequence. Previously this was determined by the ASCEND or DESCEND keyword on the array definition and SORTA used the defined sequence - which of course meant that without playing games (re-mapping the array via pointers etc.) any given array could only ever be in ascending _or_ descending order. Now the op-code extenders (A) and (D) can be used to specify the sequence in which the array is sorted.

The * is used to indicate the level at which the sorting should occur. Of course, in these examples, it's pretty obvious, since it's the only level where sorting is possible. But this sorting capability also works with nested Data Structures, so even very complicated structures can be sorted.

Searching DS Arrays

%LOOKUP can now also search DS arrays

- The same asterisk (*) notation is used to indicate the search level

Only the vanilla %LookUp is supported

- Not the %LookUpGt etc. versions

Ensures that %Lookup uses fast search

```
D productInfo     DS                  Dim(1000)
D                                     Qualified Ascend
D  productCode                  5a
D  description                 40a
D  unitPrice                    5p 2
D  qtyInStock                   5p 0

D element         S             5i 0

SortA productInfo(*).productCode;  // Sort into product sequence

// Find product code A123C
element = %LookUp( 'A123C': productInfo(*).productCode);

SortA productInfo(*).unitPrice;  // Sort into price sequence

// Locate product with unit price of $50
element = %LookUp( 50.00: productInfo(*).unitPrice);
```

Notes

To be truly useful, any enhancement in sorting needs to be matched with corresponding advances in searching, and the RPG developers haven't let us down. They have enhanced the %LOOKUP BIF to allow for searching within DS arrays.

At this time only the "exact match" %LookUp is supported. Hopefully %LookUpGt and other members of the family will be supported in future releases. In the meantime you will have to write your own routine to do this.

Notice that I have specified the Ascend keyword against the array definition. When this is used %Lookup can use a binary search which is orders of magnitude faster than the conventional linear search that would otherwise be used.

BUT ... you _must_ make certain that the array is in ascending sequence on the lookup key or you will get a lot of false misses.

Using ALIAS Names

ALIAS names can be used in Externally-described data structures

- By using the ALIAS keyword on the DS definition

ALIAS names can be used for file fields

- Use the ALIAS keyword on the File specification
  - Any LIKEREC or EXTNAME DS based on the file will use the ALIAS name

```
DDS for file CUSTFILE

     A          R CUSTREC
     A            CUSTNM         25A          ALIAS(CUSTOMER_NAME)
     A            CUSTAD         25A          ALIAS(CUSTOMER_ADDRESS)
     A            ID             10P 0

     D custDs         e ds                   ALIAS QUALIFIED
     D                                       EXTNAME(custFile)

       custDs.customer_name = 'John Smith';

       custDs.customer_address = '123 Mockingbird Lane';

       custDs.id = 12345;
```

Notes

For many, many years going all the way back to the System/38, the database has supported the use of longer alias names as an alternative to the cryptic 10 character names that we normally use. Indeed many COBOL shops have always taken advantage of them. Usage of alias names has also increased in recent years with the growth in popularity of SQL. But during all this time RPGers were locked out of using these longer names as the language was tied to the old I and O specs and their limited field names.

When result field I/O was first introduced for externally described files - back in the V5R2 timeframe - we felt that this might herald the arrival of Alias names into RPG. Well it has taken a few releases, but it is finally here. As from V7.1 you can specify the ALIAS keyword when defining an externally described DS, or any DS defined with LIKEREC. When the ALIAS keyword is used, the compiler uses the longer alias name for the field rather than the short name. In cases where the alias name does not meet RPG naming standards, the compiler reverts to using the short name.

Note that you put the ALIAS keyword on the F spec if you choose to create the DS using LIKEREC. However, if you create the DS using EXTNAME, then use specify ALIAS on the DS description on the D spec.

Simplified Rules for Data Structure I/O

Prior to these changes it was hard to use DS I/O in some cases

- For example when using READ and Write on the same file you had to have two structures - one defined *Input and one *Output
- Then use Eval-Corr to copy the data from one to the other

Now you can omit the type parameter and use the DS for all operations

- Or specify *ALL which is perhaps a more obvious approach
  - This was supported for WORKSTN files in V6
- Compiler determines which fields are input set and which are output

```
dcl-f diskf usage(*update : *output);
dcl-f prtf printer;

dcl-ds diskDs extname('DISKF' : *all) end-ds;
dcl-ds prtDs  extname('PRTF' : *all)  end-ds;

read diskfmt diskDs;
write diskfmt diskDs;
update diskfmt diskDs;

write prtfmt prtDs;
```

Notes

Data Structure I/O can be a very useful capability - but caused problems with Workstation files and any file where both input and output is required on the same file. These relaxed rules make life far simpler.

Scan and Replace

Partner400

New built-in function %SCANRPL

%SCANRPL (scanFor : replaceWith : targetString)

- Replaces all occurrences of scanFor within the target string with the contents of replaceWith
  - Optional 4th & 5th parameters for scan-start-position and scan-length
  - %SCANRPL(scanFor : replaceWith : target { : scan start { : scan length })

Much simpler than having to code it the old way

- i.e. a %SCAN and %REPLACE loop
  - Or %Scan and %Subst or ...

```
string1 = 'See &NAME. See  &NAME  run. Run  &NAME  run.';

string2 = %ScanRpl('NAME' : 'Forrest' : string1);

// string2 now contains 'See Forrest.  See  Forrest run.  Run Forrest run.'

string3 = %ScanRpl('  ' : ' ' : string2); // Change double spaces to single

// string3 now contains 'See Forrest. See Forrest run. Run Forrest run.'
```

Notes

Partner400

Note that in the example shown, replacing double spaces with a single space will only work on sets of double spaces on the first pass through the string. In other words, if there were 4 spaces in a row, the new string would still have 2 spaces. If there were 3 spaces initially there would still be 2 spaces after the replacement.

But %ScanRpl can be used with a null string as the replacement - does that help? Well it does but it creates the problem that if there were originally an even number of spaces then after the replacement there are none! Since we don't know exactly where the spaces originally were, we have no idea where to put the required space back in. Sometimes it seems you just can't win. But we have shown a possible solution on the next page - it looks odd - but it works.

Scan and Replace - Example

The code below will remove all instances of multiple spaces

- And replace them with a single space

DSPLY op-codes allow you to see it happening step by step

```
d testString      s             40a   Inz('4 spc    7 spc       10+ spc ')
d result          s             40a   Varying

 /free

  result = %ScanRpl( ' ': '<>': testString ); // replace each space with <>

  dsply ('Contents: ' + result );

  result = %ScanRpl( '><': '': result ); // replace all >< with nothing

  dsply ('Contents: ' + result );

  result = %ScanRpl( '<>': ' ': result ); // replace remaining <> with space

  dsply ('Contents: ' + result );

  dsply ('Length of result is ' + %Char(%Len(result)));
```

Notes

This approach to the problem uses a rather strange looking sequence. It really does look "odd". For that reason I have included a number of displays so that you can see the change in the string as each phase progresses. I suspect there may be a better/shorter way of doing this but this one is fun anyway. If you decide to use it in a program PLEASE wrap it up in subprocedure or you will confuse the heck out of those who follow you!

Below you can see the displays produced:

```
DSPLY  Contents: 4<>spc<><><><><>5<>spc<><><><><><>6<>spc<>
DSPLY  Contents: 4<>spc<>5<>spc<>6<>spc<>
DSPLY  Contents: 4 spc 5 spc 6 spc
DSPLY  Length of result is 18
```

V7.3 Updates

Partner400

New Op-code

- %SCANR (Scan reverse)

Works the same way as %SCAN

- Except backwards!
  - %SCANR(search argument : source string { : start position } { : length })
- Start position specifies the beginning of the substring to be searched
  - Assumed to be position 1 if not specified
- Length effectively defines a substring (i.e. from start for length)
- %SCANR starts at the end of the (sub)string and searches backwards until it finds a match or it reaches the start position

The Length parameter was also added to %Scan

  - %SCAN(search argument : source string { : start position { : length } })
- Specifies the maximum length to search within the source string

Notes

Partner400

IBM's Barbara Morris has supplied a number of examples of how to use this support - for example in the code below the file name is extracted from the full path name.

```
path = '/home/mydir/other/whatever/a.txt';
lastSlash = %SCANR('/' : path);
if lastSlash = 0;
     fileName = path;
else;
     fileName = %subst(path : lastSlash + 1);
endif;

lastSlash = 27
fileName = 'a.txt'
```

New RTNPARM Prototype Keyword

Partner400

Performance Boost for large return values

- Converts the return value to a hidden parameter
- Value reported by %PARMS( ) reflects the extra parameter
  - But it will otherwise be "invisible" to RPG callers
  - If calling from another language you will see the return value as the 1st parm
- Will improve performance when returning large values
  - Especially very large varying values

Has a second unintended use

- It allows subprocs with return values to be used by Java or stored procedure
  - Prior to this change you were limited to a 4 byte integer (10i)

```
D getFileData     pr              a   Varying Len(1000000)
D                                     RtnParm
D   File                          a   Const Varying Len(500)
D Data            S               a   Varying Len(1000)

  Data = getFileData ('/home/mydir/myfile.txt');
```

Better Support for Optional Parameters

Partner400

In the past, checked %Parms for a hard-coded value

- i.e., If %Parms > 1;

Now use %ParmNum( )

- Replaces hard-coded value for parm position
- %ParmNum returns the sequence of the parameter in the parm list

In example below, %ParmNum(Optional2) returns 2

- i.e., Optional2 is the 2nd parameter declared in the PI

```
D OptionalTest    PI
D   Parm1                       20A
D   Optional2                    5P 0 Options(*NoPass)

D Parm2           S              5P 0 Inz(99999)

  // Check for optional parameter and use value if present

  If %ParmNum(Optional2) <= %Parms;
    Parm2 = Optional2;
  EndIf;
  If Parm2 ............
```

V 7.3 Improved Null Indicator Support

SQL defined tables are appearing more frequently in our shops

- And they commonly makes use of null capable fields

Specifying ALWNULL(*USRCTL) lets you handle null values in your code

- As of V 7.3 that is now the default

A null value indicates the absence of data

- Rather than the way we have done it in the past by using special values
  - e.g. A zero in a "date" field indicated that there was no date. Zeros in the "last order number" indicated a new customer who had yet to place an order
- You should always test for null - the field value could be anything!

When a null capable record is read the null indicators are also read

- Prior to V 7.3 their value could only be tested using %NullInd (fieldName)

```
Read CustMaster;

If %nullInd( orderDate );
    // Customer has never placed an order
```

Notes

In addition to testing if the null flag is set before using a field's value, the programmer is also responsible for setting the null flag for any records being added to the database or if a value is being updated that was previously null.

For example when a customer places an order you might have logic like this:

```
orderDate = %date();
%nullind(orderDate) = *off;  // set null flag off to indicate valid data
update custRec;
```

V 7.3 Better Null Indicator Support

NULLIND keyword associates an indicator with a field

- In the example below orderDate_is_null is associated with field orderDate

Makes the testing of nullness much more obvious

- For example instead of coding:
  - If %nullInd (orderDate);
- I can now code:
  - If orderDate_is_null;
- Much cleaner and simpler

You can now define null capable fields within your program

- Not just by bringing in external definitions

```
dcl-s  aNullCapableField nullind; // %NullInd must be use to test

dcl-s  orderDate_is_null ind;
dcl-s  orderDate  date  nullind(orderDate_is_null);

If orderDate_is_null;
// Customer has never placed an order
```

Notes

Being able to define your own null capable fields is a useful feature. Instead of having to use special values for a field you can now directly associate a flag with the field to indicate that there is no value.

For example - there may be a logical difference between a total being zero and there having been no values to add to the total. Prior to this support you would have had to have had a separate flag field to differentiate the two conditions - now you can specify that the field is null capable.

You can specify NULLIND by itself - in which case %NullInd(...) must be used to test the status. Or you can associate an indicator with the field as shown in the example.

V7.3 Enhancement to LikeDS for Nulls

NULLIND can also be used with Data Structures

LIKEREC (recordName : *NULL)

- Defines a DS of null indicators for null capable fields in the record
- Subfields have the same names as the corresponding fields

Also applies to EXTNAME

- EXTNAME(file:*NULL)

```
dcl-ds custData       likerec(custRec) nullind(custData_null);
dcl-ds custData_null  likerec(custRec : *null );

      read custMaster custData;

      if custData_null.lastOrder; // Customer has never ordered
```

Notes

Free-Form In One Chart!

File definitions can be intermixed with data definitions

- Named Constants, Data Structures, etc.

Most new options have sensible defaults

- Disk files default to input, Printers to output, Decimals to zero, etc. etc.

End of line comments are now useful in definitions!

```
ctl-opt option(*srcstmt) dftactgrp(*No);

dcl-ds employeeDS;  // Nice to be able to have comments here!
  firstName char(16) Inz('James');
  lastname  char(30) Inz('Joyce');
  salary    packed(7:2) Inz(12500);
end-ds;

// Define printer file and associated DS
dcl-f qprint printer(80);     // This printer is program described
dcl-ds prtDs len(80) end-ds;

dsply ('Hello to our new employee');
dsply ( %TrimR(firstName) + ' ' + lastName );

prtDs = 'The name of our new employee is ' +
        %TrimR(firstName) + ' ' + %TrimR(lastName) +
        ' his salary is $' + %Char(salary);
write qprint prtds;
```

For full details on free-form stay in your seats for Susan's session

Note that the Printer is defined together with the DS that it uses for output

Notes

The idea of mixing file and data definitions will take some getting used to - but it makes sense.

After all it makes far more sense to define a set of variables, data structures and constants together with the file that they will be used with that to arbitrarily separate them as we had to before.

Latest DS Enhancement (V7.2 & V7.3)

RPG now allows you to directly code nested data structures

- Much easier to create nested DS
  - See the Notes page for the old fixed form version

This is a huge improvement - And proof that RFEs work!

```
dcl-ds  Customers  Qualified;
  RecordCount  Packed(3);

  dcl-ds  Customer  Dim(99);  // <<== Nested in Customers DS
    type          Char(10);
    Contact       Char(32);
    Company       Char(32);
    discountCode  Char(1);

    dcl-ds  Address;          // <<== Nested in Customer DS
      Street  Char(32);
      City    Char(20);
      State   Char(20);
      Zip     Zoned(5);
    end-ds;

  end-ds;

end-ds;
```

This field is: Customers.Customer(i).Address.Zip

Notes

The latest updates to RPG are a huge help in defining data structures for use with XML-INTO as the new syntax supports the direct coding of nested structures as you can see on this chart.

Why do I say that it is "... proof that RFEs work!" ? Because I wrote the Request For Enhancement (RFE) that resulted in this change!

This is how the same DS had to be coded prior to this latest update.

```
Dcl-Ds customers Qualified;
  recordCount  Packed(3);
  customer     LikeDS(customer_T) Dim(99);
End-Ds;

Dcl-Ds customer_T  Qualified  Template;
  type          Char(10);
  company       Char(32);
  discountCode  Char(1);
  address       LikeDS(address_T);
End-Ds;

Dcl-Ds address_T  Template;
  street  Char(32);
  city    Char(24);
  state   Char(2);
  zip     Zoned(5);
End-Ds;
```

If You Want More Columns…

If the first line of your source says **Free

- **Free must begin in line 1, column 1 (ironic, isn't it?)
- Then you can begin coding in column 1
- And continue coding all the way to the end of the source line
- Requires V7.3 or V7.2 TR3 or V7.1 TR11
- **Free members cannot contain ANY non-free-format statements
- If fixed-format code is needed, use /Copy to bring it in at compile time
  - Each /Copy member is assumed to be "column-limited" unless line 1 says **Free
- /Free and /End-Free are not only not required - they are not allowed

SQL pre-compiler supports **Free

RDi 9.5 and later supports **Free

- But not SEU, of course

Source members are limited in source lines to 32,766

- That ought to be long enough!
- IFS files are unlimited

Notes

Want New Features in RPG ?

RPG is now part of the RFE (Request for Enhancements) process.

- You can submit requirements
- You can vote on requirements that others have requested

Check out the current RFEs for the RPG compiler: ibm.biz/rpg_rfe

When searching or creating you may need to specify:

- Brand: ..... Servers and Systems Software
- Product family: Power Systems
- Product: ..... IBM i
- Component:.... Languages - RPG

To vote or submit an RFE you will need to have an IBM Id.

- Just register with an email address and away you go !

Notes

Please, please, please participate in the process. Even if you don't have any ideas of your own vote for the ones that others have submitted to help IBM prioritize.

Resources

Partner400

Articles by Jon Paris and Susan Gantner

- Search for them from the IBM Systems Magazine site
  - www.ibmsystemsmag.com/authors/Susan-Gantner/
  - www.ibmsystemsmag.com/authors/Jon-Paris/

Free-Format RPG IV: Third Edition

- by Jim Martin
- Published by MCPress (www.MC-store.com)
- Make sure to order the third edition - older versions do NOT contain the V7 free-form additions

Notes

Partner400

As noted on the chart - if buying Jim Martin's book make sure you get the third edition or later. The earlier versions do not include the V7 enhancements that we have been discussing in this session.

Any Questions ?

Partner400

?

Please e-mail me at:
Jon.Paris @ Partner400.com
for any questions