

COMMON Certification

ILE RPG Exam Study Guide

ILE RPG Certification Prep - Part 1

by Birgitta Hauser

Today, programs written in the late 80's and early 90's run on the newest IBM i release without having had to be modified in the last 20 years. But the greatest benefit may also be the downside of the programming language, RPG. For maintaining old programs, the old and meanwhile outdated methods can still be used. Since new programs can also be written in the same old style, no RPG programmer was ever forced to learn something new. And why should the management waste money for education which is not necessary?!

Currently we are faced with a situation, where RPG programmers with updated skills are desperately needed and sought after, while almost only programmers using old-fashioned coding methods are found. To make it easier for both, the hiring organizations and the applicants to test current skills, a group of the most experienced RPG programmers in the Common Community designed the certification exam for the "COMMON Certified Application Developer – ILE RPG on IBM i on POWER".

This exam does not cover RPG trivia, but is addressed to RPG programmers, with at least 3-5 years of experience in designing and writing RPG programs by using modern RPGIV programming techniques, such as: new operation codes, built-in-functions, qualified data structures, prototyping or free-format coding. Besides the core RPGIV coding techniques, the programmer should also be familiar with ILE concepts (i.e. he/she should know how to write (sub-) procedures and functions, how to create modules, programs and service programs, what activation groups and binding directories are, etc).

The exam itself is split into 4 exam areas:

- **Core RPG, Subfiles, and Externally Described Files**

This exam area covers basic RPGIV, but includes all new techniques and enhancements introduced after RPGIII until the current release. This part is focused on creating and using externally described database, display and/or printer files, defining variables, arrays and named constants within the D(Definition)-specifications, using operation codes with and without extended factor 2, using built-in-functions etc.

This article will discuss the core RPG skills required and where you can get more detailed information about the different topics and how to prepare for the exam.

- **Advanced RPG & Problem Determination / Resolution**

This exam area is focused on free-format coding, complex expressions, all kinds of qualified, overlaid, internally and externally described data structures and array data structures, prototyped program calls, error handling based on built-in-functions or monitor groups.

- **RPG Data Handling**

This exam area will cover advanced data access methods, i.e. reading and writing from/into data structures, using different methods in coding key lists, using local file access and using embedded SQL for data access.

- **ILE Concepts**

This exam area will cover the understanding and use of the ILE concepts, i.e.

Designing and writing internal and exported procedures or functions, creating modules, binding programs and service-programs, maintaining binding directories, coding and using the binder language.

Basic RPG skills required for the ILE RPG certification

RPGIV is still divided into various specifications to be coded in a predefined sequence. But with the in-

introduction of RPGIV the E and L-specifications disappeared, while new D- and P-specifications were added. We will walk through the different specifications in the sequence they have to be coded, give a short overview of the required skills and where detailed information for training and preparation can be found.

Control-Specifications / H-Specifications

In RPGIII all Control Information (H-Specs) were compressed in a single row using a column oriented and rather cryptic notation.

In RPGIV the Control information (H-Specs) are based on keywords that can be coded in any sequence between position 8 and 80 and even can be split over multiple lines. Besides the edit keywords for numeric and date values, the currency unit and the debug information already available in RPGIII multiple and very powerful keywords were introduced. Those keywords allow for example compile options to be stored within the source code. A complete list and explication for all H-Spec keywords can be found in the ILE RPG Language Reference – Chapter 12: Control Specifications.

The following list shown gives an overview of some of the most important keywords:

OPTION(*NoDebugIO)	In debug mode, no breakpoints for input and output specification are generated, i.e. after a read or chain statement the debugger continues with the next RPG command.
EXTBININT(*YES)	Integer/Binary columns in tables/physical files are not converted into packed, so the complete valid data range can be handled.
ALWNULL(*USRCTL)	NULL values can be read from and/or written to (NULL-capable) columns/fields in physical files/SQL Tables using native I/O.
DFTACTGRP(*NO)	The program is executed in an activation group other than the default activation group.
ACTGRP(XXX)	The activation group in which the program is executed. Can only be used in composition with keyword DFTACTGRP(*NO).
BNDDIR('DIR1': 'DIR2')	When compiling the object, the listed binding directories are searched to find the bound procedures.

The following code snippet shows an example of H-Specs.

```
H Debug DecEdit(0.) ExtBinInt(*YES) DATEDIT(*MDY/)
H DftActGrp(*No) ActGrp(MYACTGRP)
H BndDir(DWBNDDIR: ,WXBNNDDIR)
H AlwNull(*UsrCtl)
H Option(*NoDebugIO)
```

Example 1: H-Specifications

You may decide to store the H-Specs in either a copy member and include it with the /COPY directive into all of your programs or in a data area named RPGLEHSPEC located in your library list or a data area named DFTLEHSPEC in QR-PGLE

File-Specifications / F-Specs

Internally described database, display or printer files should no longer be used. Instead a developer should use externally described files. Externally described database files are accessed with record level access (RLA), externally described display files are needed for Green Screen applications and externally described printer files are used for spool file output.

Before defining externally described files within the F-Specs they must be defined and created. The programmer must be able to create database files with either DDS (Data Description Specification for physical and logical files) or even better with SQL (for tables, views and indexes). As an aside: SQL tables can be used in composition with native I/O like any physical file, while views can be processed like any unkeyed logical file and indexes can be accessed like any keyed logical file.

Depending on the operations to be performed, the files must be defined as read only or updatable or output files (I, U or O in position 17 in the F-Specs). For keyed access a K must be specified in position 34 while for unkeyed access this position is left *Blank.

The underlying RPG cycle automatically opens all files as soon as the program is called. However there may be some situations where a file must not be opened automatically but has to be opened manually. For example the name of the file to be accessed is determined within the program. To prevent the RPG cycle from opening a file, the USROPN keyword can be added to the F-Specs for the specific file. Within the Calculation(C)-Specifications the %OPEN() built-in-function can be used to check whether the file is already opened or not. The operation codes OPEN and CLOSE allow to control the opening and closing of user controlled opened files.

New F-Specification keywords for database files supersede the execution of additional override file commands at compile and runtime. The following list shows some of these keywords (the complete list of all keywords as well as the detailed description of the F-Specifications can be found in the ILE RPG Language Reference – Chapter 13 – File description specifications.):

EXTDESC(File)	Indicates the file to be used by the compiler to obtain the external description. Overriding database files or creating duplicates before compilation are no longer needed.
EXTFILE(File)	Specifies which table/file (in which library) must be used at runtime. Instead of hardcoding a file a variable that is set at runtime can be used.
The special value *EXTDESC indicates the file specified for compilation (keyword EXTDESC) will also be used at runtime	
EXTMBR(Member)	Specifies member is to be accessed at runtime. It can be used in composition with the EXTFILE keyword.
LIKEFILE(File)	Is used to define one file based on the definition of another file.
ALIAS	Can be used for DDS described files with fields defined with an additional alias name or for SQL described tables/indexes containing columns with long SQL names. Instead of the cryptic maximum 10 character system names, the long alias or SQL names are used.

When compiling the following source code MYFILE file in MYOBJLIB library is used to obtain the external description, but at runtime TMPFILE file in the QTEMP library will be used. The member to be accessed will be determined at runtime.

```

FMYFILE  IF  E          DISK  ExtFile(VarFile) ExtMbr(VarMbr)
F          ExtDesc(,MYOBJLIB/MYFILE')
F          UsrOpn
*-----
D VarFile      S          21A  inz(QTEMP/TMPFILE')
D VarMbr       S          10A

```

Example 2: F-Specs - Defining database files

File Specifications and Display Files

Even though third party software packages or RPG-OA (RPG Open Access) allow RPG applications to use graphical interfaces, green screen applications with DDS described display files are still commonly used.

Defining and controlling subfiles is considered as the supreme discipline in working with display files.

Before including display files in the F-specification, the display file must be designed and coded.

When defining a subfile a subfile format and a subfile control format must be created. All columns to be displayed must be listed within the subfile format, while the subfile control format must include all keywords necessary to clear, control and display the subfile.

The following list shows some of the most important keywords used for controlling subfiles. The complete list of all DDS keywords can be found in DDS for display files – DDS Keywords for display files (see link below).

SFLPAG	Specifies the number of records to be displayed at the same time on the screen
SFLSIZ	Record-level keyword on the subfile control format to define the number of records in the subfile
If SFLPAG and SFLSIZE are identical, a page at time subfile is defined. Rollup and Rolldown must be controlled manually.	
If SFLSIZE is greater than SFLPAGE a self-extending subfile is created.	
SFLCLR	Removes all subfile records from your subfile
SFLDSP	Displays the subfile records if an output operation is sent to the subfile-control record format
SFLDSPCTL	Displays the subfile control format
SFLEND	Controls whether +, More... or End is displayed on the end of the subfile to indicate if the last record is reached or not.
SFLCSRNRN	Returns the relative record number of the record on which the cursor is located within a subfile.
SFLRCDNBR(Cursor)	Is used to specify that the page of the subfile to be displayed is the page that contains the record whose relative record number is in this field.

For all display files (independent of it being a subfile or an ordinary input/output screen) function keys must be defined and associated. In your RPG program you need to be able to check which function key is pressed and perform the appropriate action.

Within the RPG program each subfile format and the associated subfile row counter must be added to the display file definition within the F-Specifications by using the keyword SFILE. Before writing any record into a subfile, the subfile must be initialized/cleared first, i.e. the indicator associated with the SFLCLR keyword must be set *ON, the subfile control format must be written through the WRITE operation code and the SFLCLR indicator must be set *OFF after.

For each row to be displayed in the subfile the columns defined within the subfile format must be filled with data first and the subfile row counter must be raised by 1. Then the subfile format must be written by using the RPG operation code WRITE. After all subfile records that are to be displayed are written, the indicators for the keywords SFLDSP and SFLDSPCTL must be set and the subfile control format is sent out by using the EXTFMT (Write/Then Read Format) Operation Code.

A series of articles about subfile handling can be found on the IT Jungle – The Four Hundred Guru website (see link below).

Definition Specifications / D-Specs – Defining standalone Fields and Arrays

The D-Specifications allow all kinds of variables, arrays, data structures, data areas and constants but also prototypes and procedure interfaces to be defined. Even though the D-Specifications are column oriented, only a single format is used for all definition types.

The definition type specified in position 24-25 determines what type of item is to be defined. The definition type for a standalone field is S, while it is C for a constant, DS defines the data structure header specification, PR is the main line for a prototype and PI is the main line for a procedure interface. For Data structure subfields or parameter in a prototype or procedure interface the definition type is left blank.

RPGIV does not only support packed, zoned and binary numeric variables, but also signed and unsigned integer variables, float and double variables. Integer variables are defined with the data type I and a length of 3 (= 1 Byte), 5 (= 2 Byte), 10 (=4 Byte) and 20 (= 8 Byte). Unsigned integer variables are defined with the data type U and the same length specifications that has to be used for integer variables. While integer variables can hold negative and positive values, unsigned integer variable values must always be greater or equal *Zeros. Contrary to binary variables signed and unsigned integer variables are not converted into packed data. Float and double variables are both defined with data type F but

float has a length specification of 4 while double has a length specification of 8.

There are 3 alphanumeric data types. Single Byte Character variables are defined with data type A. Double byte character variables can be either defined with data type C (= UCS-2 format) or G (= Graphic) for any other double byte character set. With release 6.1 the maximum length for character variables is increased to 16 MB. Unicode variables can be defined up to a maximum length of 8MB characters (1 Character = 2 Byte). To define a character variable with the maximum length, the LEN keyword must be specified. With the VARYING keyword varying length, instead of fixed length, variables can be defined.

For indicators a special character data type N was introduced. An indicator variable is defined as character variable with a length of 1 byte but can only hold two different values, *ON and *OFF.

Date variables (not numeric dates!) are defined with the data type D, without any length specification. The length is automatically determined based on the date format to be used. The date format can be set through the keyword DATFMT in composition with a valid date format for example *ISO, *EUR or *YMD. If the keyword DATFMT is not specified in the D-Specification, the date format specified in keyword DATFMT in the H-Specs is used. If no date format is specified, neither in the D- nor in the H-Specs, date format *ISO (YYYY-MM-DD) is used.

Time variables are defined with the data type T without any length specification. The time format, to be used, can be pre-defined by specifying the keyword TIMFMT in composition with a valid time format, for example *HMS or *USA.

Timestamp variables are defined with data type Z without any length specification.

The data type pointer (*) was originally introduced to call functions in written in other languages such as C. With built-in-functions (for example %ADDR() or %ALLOC()) and pointer arithmetic, storage can be reserved, freed or the pointer set to a different address.

To be able to call JAVA methods the data type O (=Object) was introduced. When defining an object variable the keyword CLASS with the qualified class name must be specified.

For additional information about the data types allowed in RPGIV please refer to the ILE-RPG Language Reference – Chapter 9 (see link below).

All variables can be defined by a data type and if necessary with a length specification, but it is also possible to define a variable based on the definition of another variable simply by using the LIKE(RefFld) keyword in composition with the reference variable. By specifying +X or -X in the length specification it is possible to define the new variable smaller or larger than the reference variable.

To initialize the variables with a specific value, the keyword INZ with an initial value can be specified. Instead of hardcoding an initial value a couple of figurative constants can be specified. For example *LOVAL or *HIVAL will set the initial value to the minimum or maximum value depending on the data type and length of the variable. The figurative constant *SYS initializes a date field with the system date (=Current date), while *JOB initializes the date field with the job date (the date when the job started, which may or may not be identical to the system date).

Global variables are only initialized the first time the program is called, while local variables are initialized each time the procedure is called, except for variables defined with the STATIC keyword.

In the following example a couple of variables with different data types, lengths and initial values are defined.

D Quantity	S	11P 3
D Total	S	+2 like(GblQuantity)
D ACT	S	3 inz(,ABC')
D FirstRow	S	N inz(*On)
D Integer	S	10I 0 inz(*Loval)
D Unsigned	S	5U 0 inz(*Hival)
D DateIso	S	D DatFmt(*Iso) inz(*SYS)
D DateEur	S	D DatFmt(*Eur) inz(D'2002-01-01')
D TimeHMS	S	T TimFmt(*HMS:)
D TimeStamp	S	Z

Example 3: D-Specifications - Defining standalone fields

Arrays can be defined as standalone fields or data structure subfields, by adding the keyword DIM(NbrElem) to the variable definition. Descending and ascending sequence of the array elements can be predefined by adding either the ASCEND or the DESCEND keyword. When defining an array using the LIKE keyword, the DIM keyword is not inherited, i.e. it must be specified for each array.

The number of elements can be specified as named constant or as return value of a built-in-function, such as %LEN(), %SIZE() or %ELEM(). For defining an array with the same number of elements as another array, the built-in-function %Elem(RefArray) can be used.

The following example shows different arrays defined based on a data type and length or on a reference variable. The number of elements is either hardcoded or determined at compile time through a built-in-function.

```
D ArrXX      S      11P 3 dim(15) Descend

D Text       S      50
D ArrWord    S      50  Dim(%len(Text)) Varying
*
D ConstMaxArr C      const(1000)
D ArrDisplay S      like(Status) dim(ConstMaxArr)
D ArrProgram S      like(Status) dim(%elem(ArrDisplay))
D ArrDescr   S      like(Text)  dim(%elem(ArrDisplay))
```

Example 4: D-Specifications - Defining standalone arrays

Named constants are defined with the definition type C. Specifying the CONST keyword for defining the constant value is optional. The constant value can also be determined through a couple of built-in-functions that can be specified within the D-Specs.

In the following code snippet named constants are defined based on a hardcoded value or using built-in-functions.

```
D ConstMaxArr C      const(1000)
D ConstLower  C      ,abcdefghijklmnopqrstuvwxyz'
D ConstUC2Blank C      Const(%UCS2(, ))

D Decimal     S      10P 3
D ConstDecLen C      Const(%Size(Decimal))
D ConstDecPos C      Const(%DecPos(Decimal))
```

Example 5: D-Specifications - Defining Named Constants

Calculation-Specification / C-specs – Operation Codes

Within the Calculation Specification (C-Specs) data is read and modified through operations that can be either performed by using an operation code or a built-in function.

All operation codes that were available in RPGIII can be used in RPGIV fixed format coding in the same way, while in free format coding not all of these operation codes are supported. But there are alternatives such as new operation codes, operation codes with an extended factor 2 and built-in-functions.

This exam perspective is focused on the new and enhanced operation codes.

Calculation-Specification – Assignment Operation Codes

The traditional MOVE, MOVEL, MOVEA, Z-ADD, ADD, Z-SUB, SUB, MULT, DIV operation codes can be replaced by the EVAL (Evaluate expression) operation code that evaluates an assignment statement of the form “result = expression”.

The result can be any standalone variable, array, array element, data structure or data structure sub-field defined in any RPG data type. For arithmetic expressions the + (addition), - (subtraction), / (division), * (multiplication) and ** (power) mathematical operators can be used as well as (nested) parenthesis.

Assignment operators +=, -=, *=, /=, **= allow a variable to be modified based on its old value, for example NextSeq += 1 (Add 1 to NextSeq).

When using the EVAL operation code in composition with character expressions, multiple character variables, literals and constants can be concatenated by using a simple + sign.

The operation code EVALR (Evaluate expression, right-adjust) allows the result of character expressions to be placed right adjusted in the result variable. In case the result of the expression is shorter than the fixed length result variable, it is padded with blanks, contrary to the MOVE operation codes where by default only the number of the moved characters is replaced.

Built-in-functions can be specified within an expression. When using variables defined with different data types (for example character and packed numeric) built-in-functions have to be used to convert the variables into compatible data types.

In RPG free format, the operation code EVAL is only required if an extender (for example (H) for rounding) is specified.

In the following code snippet the evaluation operation codes are used to execute arithmetic calculations and concatenating strings.

```
C      eval    Total = ((Var1+Var2) * (Var2-Var4))/Var4
C      Eval    Text = ,Total for Customer , + CustName +
C              , in , + %Char(lastyear) + , = , +
C              %Char(Total)
/Free
    EvalR CharTotal = %TrimR(%Char(Total) + , Eur ,;
    Total2 = Quantity * Price;
/End-Free
```

Example 6: C-Specifications - Evaluation Operation Codes

More detailed information about the Evaluation Operation Codes can be found in the ILE RPG Language Reference - Chapter 22: Operation Codes. Expressions are described in Chapter 20: Expressions (see link below).

Calculation-Specification – Structured Programming Operation Codes

According to Wikipedia structured programming is a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of subroutines (EXSR, BEGSR/ENDSR), (sub-)procedures, block structures (IF/ELSE or SELECT/WHEN) and loops (DOU, DOW, FOR).

The programmer should be able to split a task into smaller tasks and single steps and design the flow of the program.

Like the evaluation operation codes (EVAL/EVALR), all (new/enhanced) structured programming operation codes are designed with an extended Factor 2. The expression, i.e. conditions that can be specified within in the extended factor 2 can include, arithmetic operators, parenthesis, comparison operators (=, >, <, <>, >=, <=) and logical operators (*AND, *OR, *NOT).

The following list shows the most important structured programming operation codes:

IF/ELSEIF/ELSE/ENDIF	IF condition to only execute several operations if the specified condition is satisfied. An IF-Group must always be ended with an ENDIF statement.
SELECT/WHEN/OTHER/ENDSL	SELECT condition to execute several alternative sequences of operations depending on the specified conditions. A SELECT-Group must always be ended with an ENDSL

statement.

DOW/ENDDO

The DOW (Do While) operation code allows a group of operations/statements to be executed multiple times. The condition specified in the DOW statement is checked at the beginning of the loop. The loop is left as soon as the checked condition is no longer true. If the condition is not true the first time the DOW statement is executed, the operations within the loop are not executed. A DOW-loop must always be ended with an ENDDO statement.

DOU/ENDDO

The DOU (Do Until) operation code allows a group of operations to be executed multiple times. The condition specified in the DOU row is not checked before the end of the loop. The loop is left as soon as the condition is no longer true. If the condition is not true the first time the DOU statement is passed, the operations within the loop are nevertheless executed. The loop is left after the first execution if the condition is still not true. A DOU-loop must always be ended with an ENDDO statement.

FOR/ENDFOR

The FOR-operation code allows the repetitive processing of a group of calculations. A starting value is assigned to an index variable. Increment and limit values can be specified. The loop is left as soon as the limit is reached or exceeded. A FOR-loop must always be ended with an ENDFOR statement.

ITER

Can only be specified within a DOW, DOU or FOR loop to force the next iteration to be executed.

LEAVE

Can only be specified within a DOW, DOU or FOR loop to force the loop to be left immediately.

All structured operation codes (except ITER and LEAVE) can be nested.

The following code snippet shows the structured operation codes in a program flow.

```
C      DOW    Not FktKey03 and Not FktKeyF12;
C      DOW    Not %EOF(MyFileP)
C      EndDo

C      IF     ReLoad;
C      Iter
C      EndIf

C      EXTGMT MyDspFF

C      Select
C      When   FktKey04
C      When   FktKey07
C      Other
C      EndSL

C      EndDo
```

Example 7: C-Specifications - Structured Programming Operation Codes

More detailed information about the structured programming operation codes can be found in the ILE RPG Language Reference - Chapter 22: Operation Codes (see link below).

Calculation-Specification – File Operations

With RPG cycle a file can be opened, read record by record (from the first to the last) and depending on the definition

updated. In most RPG programs however only a subset of all records in a physical file/table are needed and must be accessed specifically.

For reading only a subset of data, the file pointer must be positioned at the beginning of the data set by using either the SETLL or SETGT operation code in composition with a single key or a key list. Instead of positioning the pointer the first record can be read with the CHAIN Operation Code. All records to be handled can be read sequentially with a READ operation code within a loop.

The following list shows the most important operation codes to be used in composition with record level access (RLA). A list of all Operation codes can be found in the ILE RPG Programming Reference – Chapter 22.

CHAIN	Reads a single record from a database file based on a single or compound key or the relative record no.
READ	Reads the next record from the database file
READE	Reads the next record from the database file with the specified key
READP	Read the previous record from the database file
READPE	Read the previous record from the database file with the specified key
SETLL	Positions before the first record in the database file with the specified key or relative record no
SETGT	Positions after the last record in the database file with the specified key or relative record no

When looping through the data set, it must be checked if the last record is reached or if additional records can be read. Whether a record is found or whether the last record is read can be checked through an indicator that is associated with the READ or CHAIN operation code. But instead of using indicators RPGIV provides a couple of built-in-functions that allow checking if a record is found or if an error is returned.

The following list shows the built-in-functions set in composition with record level access:

%EOF(File)	The %EOF indicator is set to *ON if the last record of the file or the data set is read. The Built-in-Function can be checked for all READ operation codes.
%FOUND(File)	The %FOUND indicator is set to on if a record is read in a CHAIN operation. It is also set to *ON in composition with the SETLL operation code if a sub-sequent read would be successful, but it does not indicate that a record with the specified key is found.
%EQUAL(File)	The %EQUAL indicator is set to *ON in composition with the SETLL operation code is found, if a record with the specified key is found. In this way %FOUND may be set to *ON while %EQUAL is set to *OFF.
%ERROR	The %ERROR indicator is set to *ON if the extender (E) was specified with the operation code and the operation failed.
%STATUS	The %STATUS built-in-functions is also set in composition with an (E) extender in the operation code in case the operation fails, but it delivers more detailed information than the %ERROR indicator.

Records in files or tables are not only read but also written, updated or deleted by using the appropriate operation codes:

WRITE	Write a new record into the database file
UPDATE	Update the last locked record retrieved from the database file
DELETE	Deletes a record from a database file

In the following example the first record based on a (classical) key list is read. No Indicator is associated with the chain statement, but the %FOUND built-in-function is used to check whether a record was read or not. If a record is found it is processed and updated within the DOU (Dountil) Loop. At the end of the loop, immediately before the ENDDO operation the next record is read. In a DoUntil loop the condition is checked at the EndDo. If no further record is found the loop is

left.

```
C   KList01   Chain   MyFileF

***** Record Found
C       If      %Found(MyFileP)

C       DOU      %EOF(MyFileP)
C***** Do whatever you want
C       UPDATE   MyFILEF

C   KList01   READE   MyFileF
C       EndDo

C       EndIf
```

Example 8: C-Specifications - Record Level Access

Calculation-Specification – Built-in-Functions

Built-in functions are similar to operation codes in that they perform operations on data. Built-in functions can be used within expressions.

Unlike operation codes, built-in functions return a value rather than placing a value in a result field.

Built-in-Functions can be specified in an expression that is specified with the evaluation operation codes or in the extended factor 2 of a structured programming operation code. Because built-in-function returns a value, it also can be specified as operand in another built-in-function.

The RPGIV programming language currently provides around 80 built-in-functions, for all kinds of operations. For examples built-in-functions for string manipulation (%TRIM, %SUBST), date and time calculation (%DATE, %DIFF), file access (%EOF, %FOUND), error handling (%ERROR, %STATUS), data type conversion (%CHAR, %DEC). There are also built-in-functions that can replace or enhance operation codes, for example %CHECK (instead of CHECK), %SCAN (instead of SCAN) or %LOOKUPxx (instead of LOOKUP).

The following example shows the use of some built-in-functions. In the first statement *Blanks are removed from the first and last name by using the %TRIM and %TRIML Built-in-Functions. Both revised names are concatenated separated by a blank. To calculate the Total, USD and all commas are converted into blanks by using the built-in-function %XLATE(). The result is converted into a numeric value by using the built-in-function %DECH(). In the free format example, all – (dashes) are removed from the string. With built-in-function %SCAN the position of the next dash is determined and with built-in-function %REPLACE this dash is removed. Beginning with release 7.1 a do loop is no longer needed, since the new built-in-function %SCANRPL can replace all occurrences.

```
C       Eval    FullName = %Trim(LastName) + ' ' +
C               %TrimL(FirstName)

C       Eval    CharTotal = ,USD 12,456,789.25 ,
C       Eval    Total = %DecH(%Xlate(,USD,' , , ,: CharTotal):
C               15: 2)
/Free
Text50 = ,ABC-DEFGHI-J-KLM-NOPQR';
DoW %Scan(,-': Text50) <> *Zeros;
    Text50 = %Replace(,' : Text50: %Scan(,-': Text50): 1);
EndDo;
/End-Free
```

Example 9: C-Specifications - Using Built-In-Functions

Calculation-Specification – Date and Time Calculation

The RPGIV language provides several operation codes and built-in-functions, for converting numeric and character date/time representations into real date/time values and vice versa, for determining a portion of date or time value, for adding or subtracting date and time values and calculating time differences in a specified date or time unit.

ADDUR Adds the duration specified in factor 2 to a date, time or timestamp value.

SUBDUR Subtracts the duration specified in factor 2 from a date, time or timestamp value.

The SUBDUR operation can also be used to calculate the duration between two dates, a date and a timestamp, two times, a time and a timestamp or two timestamps.

EXTRCT Returns the year, month, day portion of a date or timestamp, the hour, minute or second portion of a time or timestamp or the microsecond portion of a timestamp

In the following example 2 hours are added to the NewTime variable, while 30 minutes are subtracted from the NewTime variable and the result is returned in the NextTime variable. With the last SUBDUR operation, the difference in Minutes between two timestamps is calculated.

```
C      Eval   NewTime = T'18.00.00'
C      adddur 2:*Hours   NewTime

C      Eval   DiffMin = 30
C      NewTime subdur DiffMin:*MN NextTime

C      Eval   MyTimestamp1 = Z'2012-12-02-14.26.45.000000'
C      Eval   MyTimestamp2 = Z'2012-12-02-10.10.10.000000'
C      MyTimestamp1 SubDur MyTimestamp2 DiffMin:*MN
```

Example 10: C-Specifications - Date/Time Operation Codes

Unfortunately those date and time calculation operation codes are not supported in RPGIV free format coding, instead alternate built-in-functions can be used.

The following list shows the built-in-functions to be used in composition with date and time calculation

%DATE() Converts a numeric or character representation of a date into a real date. The format of the character date representation can be specified in the second parameter.

If no parameter is specified with the built-in-function the current date (=system date) is returned.

To determine the job date the special value *JOB can be specified as the first parameter.

%TIME() Converts a numeric or character representation of a time into a real time. The format of the character date representation can be specified in the second parameter.

If no parameter is specified with the built-in-function the current time is returned.

%TIMESTAMP() Converts a numeric or character representation of a timestamp into a real timestamp.

If no parameter is specified with the built-in-function the current timestamp is returned.

%YEARS(), %MONTHS(), %DAYS(), %HOURS(), %MINUTES(), %SECONDS(), %MSECONDS() Those built-in-functions convert a numeric value into a time unit to be added to a real date, a real time or a real timestamp.

%DIFF() Calculates the difference between two date, time or timestamp values in the specified date or time unit, i.e. days, hours, seconds.

%SUBDT() Returns the portion of the specified time unit out of a date, time or timestamp.

%CHAR()	Converts a date, time or timestamp into a character representation of a date, time or timestamp. The date/time format in which the character representation should be returned can be specified in the second parameter.
%DEC()	Converts a date, time or timestamp into a numeric representation of a date, time or timestamp. The date/time format in which the numeric representation should be returned can be specified in the second parameter.

In the following example first the job date and the current timestamp (system timestamp) determined. To calculate the month end, the numeric date is converted into a real date and 1 month is added. The day of month - number of days of the date in the next month is subtracted. The month end date is converted into a numeric date with the built-in-function **%DEC**.

```

/Free
JobDate   = %Date(*Job);    //Job Date
SystemTS  = %Timestamp();   //Current Timestamp

DateNum   = 20121202;
MonthEnd  = %Date(DateNum) + %Months(1)
           - %Days(%SubDt(%Date(DateNum): *Days));
MonthEndNum = %Dec(MonthEnd: *ISO); //Convert date into numeric date

```

Example 11: Calculation Specification - Date/Time built-in-functions

Detailed information about date and time formats, duration codes and the valid ranges for date and time values can be found in the ILE – RPG Language Reference – Chapter 9 – Data types and data formats. Additional information about Date and Time Operations can be found in Chapter 19 – Operation. Built-in-Function for Date and Time Calculations can be found in Chapter 21 – Built-in-Functions.

Additional Links, Tutorials and Articles

In order to prepare for the exam you may find a lot of valuable information under the following links:

- ILE RPG Language Reference

<http://publib.boulder.ibm.com/infocenter/iseres/v7r1m0/index.jsp?topic=%2Frzahg%2Frzahgrpglangref.htm>

- Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More

<http://www.redbooks.ibm.com/abstracts/sg245402.html?Open>

This redbook provides among other useful information a really good and easy to understand introduction into the ILE concepts and embedded SQL.

- ILE Concepts

<http://publib.boulder.ibm.com/infocenter/iseres/v7r1m0/index.jsp?topic=%2Frzahg%2Frzahgileconcept.htm>

- Modernizing IBM eServer iSeries Application Data Access - A Roadmap Cornerstone

<http://www.redbooks.ibm.com/abstracts/sg246393.html?Open>

In Chapter 7 of this redbook you'll find an easy to understand introduction into embedded SQL.

- DeveloperWorks – RPG Café

<https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=b542d3ac-0785-4b6f-8e53-f72051460822>

Barbara Morris just recently published the first five chapters of an RPG-Beginner's Tutorial.

- IBMSystems Magazine – IBM i – Developer - RPG

<http://www.ibmsystemsmag.com/ibmi/developer/rpg/>

You'll find a lot of articles written by Jon Paris, Susan Gantner, Aaron Bartell and others

- IT Jungle – The Four Hundred Guru

<http://www.itjungle.com/fhg/fhgindex.html>

Among others you'll find a series of articles about coding and handling subfiles.

- MC-Press Online – Programming - RPG

<http://www.mcpressonline.com/programming/rpg/>

- iPro Developer (formerly System iNetwork)

<http://www.iprodeveloper.com/archives/magazines/ipro-developer>

iPro Developer provides a magazine and several newsletters. You need a subscription and not all articles are free of charge.

- Scott Klement's Webpage

<http://www.scottklement.com/>

- DDS for physical and logical files

<http://publib.boulder.ibm.com/infocenter/iseri/v7r1m0/index.jsp?topic=%2Frzskb%2Fkickoff.htm>

- DDS for display files

<http://publib.boulder.ibm.com/infocenter/iseri/v7r1m0/index.jsp?topic=%2Frzskc%2Fkickoff.htm>

- DDS for printer files

<http://publib.boulder.ibm.com/infocenter/iseri/v7r1m0/index.jsp?topic=%2Frzskd%2Fkickoff.htm>

IBM i Access for Training

If you need access to an IBM i for training, writing your own example programs, testing purposes and preparing for the exam, you may contact:

RZKH – Rechenzentrum Kreuznach (Holger Scherer)

RZKH offers free (Release V5R3) or paid (higher releases) access to an IBM i system.

<http://www.rzkh.de/cgi-bin/db2www/getaccounteng.d2w/main>

And now have fun in preparing for the ILE RPG certification exam!

About the Author:



Birgitta Hauser has been a Software Engineer since 2008, focusing on RPG, SQL and Web development on System i/Power i at Toolmaker GmbH in Germany. She graduated with a business economics diploma, and started programming on the AS/400 in 1992. Before joining Toolmaker, she was responsible for the complete RPG, ILE, and Database programming conceptions for a software house with its own Warehouse Management Software Package. She also works in consulting and education as a trainer for RPG and SQL developers. Since 2002 she has frequently spoken at the COMMON User Groups in Germany, Switzerland and USA. In addition, she is co-author of two redbooks and also the author of several papers focusing on RPG and SQL for IBM Developer Works and a German magazine.

ILE RPG Certification Prep - Part 2

by Laura Ubelhor

Business programming using RPG as a programming language continues to have an important role on the IBM i platform. We have seen introduction to new technologies over the years, but still have billions of lines of RPG code providing a backbone for many applications within both large and small organizations. RPG as a language has evolved greatly. Throughout the years, additional functionality and capability has been added resulting in a robust and powerful language for application coding and development. Today's IBM i shop benefits from development staff being able to utilize and integrate use of the advanced capabilities of the modern RPG language.

COMMON CERTIFICATION

The COMMON ILE RPG certification provides an advantage to individuals and organizations alike, as a developer earning a RPG certification shows employers your level of capability to use modern coding techniques and skills. In today's competitive business environment it is more necessary than ever to effectively deliver technology in a way that makes the most efficient use of the limited resources available in any enterprise, large or small. As an employer, an RPG certification provides a means to easily recognize a skilled candidate.

This certification exam is intended for those application developers of sufficient depth and breadth of experience who seek a credential to validate those skills. The minimally qualified candidate will possess at least:

- 3 – 5 years performing a broad range of RPG design and programming activities
- Advanced knowledge of keywords and their functions
- At least 18 months to 2 years of experience using ILE and free-format techniques

Review the exam objectives in conjunction with the certification description and prerequisites/ experience required for the exam to help you decide if the exam is appropriate for your expertise, skill, and experience.

To prepare for the certification exam, review the exam objectives as a tool to help you understand the scope of the exam and the topic areas you may need to study. Preparing for the exam is key. The following includes information on each of the areas of focus on the exam within the Advanced RPG & Problem Determination/Resolution area of the exam. In addition review the IBM ILE RPG Language Reference manual and other reference material. Studying and practicing techniques will help a test taker prepare. The areas covered within this section should not be considered all inclusive. Some additional reference materials can be found at the end of this docume

Advanced RPG & Problem Determination/Resolution

The exam is split into 4 exam areas. The second area is Advanced RPG & Problem Determination/Resolution. This exam area is focused on free-format coding, complex expressions, all kinds of qualified, overlaid, internally and externally described data structures and array data structures, prototyped program calls, error handling based on built-in-functions or monitor groups. It is strongly recommended prior to taking the exam you prepare. An understanding of the functionality and use of RPG ILE is key to being able to earn certification.

Given an Example of Complex Logical Expression, Determine its Results

An experienced programmer needs to be able to code complex logic and understand coding techniques available to provide accurate results. RPG has evolved to allow for longer field names, field length and decimal entries. Using free-format expressions can simplify the code for complex logical comparisons and calculations and also make the code more readable. What may have taken several lines of code can often be simplified into one code statement using free-format. No longer do you concern yourself with getting everything into the right columns instead you need to abide to code fitting within columns 8 to 80, leaving 6 and 7 blank. Free-format also features elimination of the use of indicator columns. Conditioning indicators, resulting indicators, and control level indicators are not allowed. The code is similar except each line starts with an operation code (instead of factor 1). Expressions are used in RPG programs for declaration, assignment, comparison or with procedures. Expressions may be coded within statements including CALLP, DOU, DOW, EVAL, EVALR, FOR, IF, RETURN and WHEN. Expressions may also make use of operands and operators. To be able to evaluate a complex expression a clear understanding of operation precedence and order of evaluation is required. A complex expression may also include use of expression operators including unary operations, binary operations, built in functions (BIFs) and user defined functions (a prototype procedure that returns a value).

It's not an uncommon task for a developer to need to write complex logic for a new development project, enhance existing code that includes complex logic or troubleshoot and debug code issues as a result of complex logic that is not coded properly. All of these tasks require a programmer to be able to read and understand logical expressions code.

```
D fielda    S      10i 0 inz(5)
D fieldx    S      10i 0
```

```
for fieldx = 1 to 3;
  fielda = fielda +field x;
  if (fielda > 6);
    fielda = fielda + 2;
  endif;
```

Results of expression

fielda = 36 and fieldx = 7

Figure 1: Complex Logical Expression

For more information on Logical expressions see the IBM ILE Reference guide Section Operations, Expressions and Functions.

Using Data Structure Arrays

Today a developer is able to make use of data structure arrays. The array is placed within a named data structure. A data structure is a grouping of related fields, defined and grouped together with an assigned name. An array data structure is like a multiple occurrence data structure, with the exception that the index is explicitly specified. An array data structure is defined using the keyword Dim as a definition keyword of a data structure.

If you have a data structure array named GLArray that is defined with 48 elements, GLArray(18) will access the 18th element of the data structure array. The qualified keyword is used to specify the name of a data structure subfield qualified by the name of the data structure. For example, if you have a data structure array named GLArray and a subfield named GLYear, then GLArray.GLYear can be used to access the subfield. By using the Dim keyword, it provides the ability to access subfields in a specific example, for example GLArray(18). GLYear provides access to the 18th element in the data structure array GLArray.

Data structure arrays are a very useful tool for a programmer. Data structure arrays may be used in place of a temporary work file without requiring creation of a file or requiring file I/O. Arrays may be used as a passed parameter allowing you

to be able to pass a large amount of organized data between sub-procedures or programs. Data structure arrays can be used as one large string of data or individual elements can be used and manipulated. A data structure array is similar to a multidimensional array in providing ability to add another dimension to an array; however, they are simpler to code.

DGLArray	DS		Qualified Dim(48)
----------	----	--	-------------------

D GLYear	4 0
D GLMonth	2 0
D GLAccount	11A
D GLCreditAmt	15 2
D GLDebitAmt	15 2

/Free

```
GLArray(1).GLYear = 2013 ;
GLArray(1).GLMonth = 01 ;
GLArray(1).GLAccount = '4012-52-100' ;
GLArray(1).GLCreditAmt = 1000 ;
GLArray(1).GLDebitAmt=10000 ;
```

/End-Free

Figure 2 – Example of data structure array

Adding another dimension to a data structure array requires adding a Dim keyword to a subfield of the data structure array. In the GLArray example the DS definition indicates there are 48 data structure elements or in this example 4 years of GL Data. To reference an element of the data structure array an index can be specified after the data structure name and before the qualifier subfield. The data structure array may also contain an array, which makes the array dimensional.

DGLArray	ds		Dim(48) Qualified
----------	----	--	-------------------

D GLYear	4 0	Dim(4)
D GLMonth	2 0	
D GLAccount	11A	
D GLCreditAmt	15 2	
D GLDebitAmt	15 2	

/Free

```
GLArray(1).GLMonth = GLArray(2).GLMonth;
GLArray(1).GLYear(1) = GLArray(2).GLYear(2);
```

Figure 3: A data structure array

The %SUBARR (Set/Get portion of an array) BIF allows you to reference a subset of the elements in an array. %SUBARR provides a means of referring only to relevant elements in an array. This is useful when used with SORTA. SORTA may not be used to sort a data structure array.

```
D Codes      S      10a Dim(100) Ascend
D noOfElements S      10i 0
```

```
/Free
```

```
SortA %SubArr(Codes:1:noOfElements);
```

Figure 4: Using SORTA and %SUBARR

We have just touched upon the basics of a data structure array. In addition a modern developer should be familiar with additional functionality available for use with data structure arrays including but not limited to:

- Keyed array data structure – an array data structure with one subfield identified as the search or sort key.
- CLEAR keyword – Use of CLEAR operation code to clear fields.
- Overlay keyword – an array can be defined within a data structure and the name of the data structure on the Overlay keyword.
- %SUBBARR – built in function (BIF) providing ability to reference a subset of the elements in an array. %SUBBARR provides a means of referencing. This is useful in coordination with the SORTA(Sort Array) operation.
- Passing data using a data structure array
- Sorting data structure arrays
- Using data structure arrays with subfiles

Code Complex D-specs (e.g., OVERLAY, coding fields without attributes, etc.)

D-specs or definition specifications can be used to define standalone fields, named constants, data structures and their subfields, prototypes, procedure interface and prototype parameters. D-specs can be placed within a program or module in the main source section or in a subprocedure. Within the main source section global definitions are defined. Within a subprocedure, the procedure interface and parameters are defined for a prototype. Local data items may also be defined for use of prototyped procedure processing. Definitions within a prototyped procedure are local. Once local definitions are defined they can be used by any other procedure including the main procedure.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
```

```
DName+++++++ETDsFrom+++To/L+++IDc.Keywords+++++++Commen
ts+++++++
```

Figure 5 – D-spec layout

- The definition specification type (D) is entered in position 6.
- The non-commentary part of the specification extends from position 7 to position 80
- The fixed-format entries extend from positions 7 to 42
- The keyword entries extend from positions 44 to 80
- The comments section of the specification extends from position 81 to position 100.
- Positions 44 to 80 are provided for definition specification keywords. Keywords are used to describe and define data and its attributes. Use this area to specify any keywords necessary to fully define the field.

Following is a list of keywords available for use with D-specs.

ALIGN	EXPORT{(external_name)}	OPDESC
ALT(array_name)	EXTFLD(field_name)	OPTIONS(*NOPASS *OMIT *VAR-SIZE *STRING *RIGHTADJ)
ALTSEQ(*NONE)	EXTFMT(code)	OVERLAY(name{:pos *NEXT})
ASCEND	EXTNAME(file_name{:format_name})	PACKEVEN
BASED(basing_pointer_name)	EXTPGM(name)	PERRCD(numeric_constant)
CCSID(number *DFT)	EXTPROC({*CL *CWIDEN *CNOWIDEN {*JAVA:class-name:}}name)	PREFIX(prefix{:nbr_of_char_replaced})
CLASS(*JAVA:class-name)	FROMFILE(file_name)	PROCPTR
CONST{(constant)}	IMPORT{(external_name)}	QUALIFIED
CTDATA	INZ{(initial value)}	STATIC
DATFMT(format{separator})	LIKE(name)	TIMFMT(format{separator})
DESCEND	LIKEDS(data_structure_name)	TOFILE(file_name)
DIM(numeric_constant)	NOOPT	VALUE
DTAARA{(data_area_name)}	OCCURS(numeric_constant)	VARYING

Figure 6 - D-spec keywords

Older versions of RPG used input specifications (I-specs) to define data elements within a program; this has been replaced with use of D-specs. I specs are still used for their original intent to define program described input files. All other data elements may be defined using D-specs. D-specs offer much more functionality and are a preferred method of data element definition. Introduction of D-specs also eliminated the need to use E-specs to define program tables and arrays.

Making use of complex D-specs including OVERLAY keyword and coding fields without attributes are techniques that should be understood and used by the modern RPG developer. Following are some examples of complex D-specs.

D DataStruct1	DS	QUALIFIED
D PartNumber	15A	INZ('13CHRYEXP12345A')
D ModelYear	2A	OVERLAY(PartNumber)
D Platform	4A	OVERLAY(PartNumber:3)
D Model	3A	OVERLAY(PartNumber:7)
D EngrNum	6A	OVERLAY(PartNumber:10)
D InternalPart	9A	OVERLAY(EngrNum:1)

D DataStruct1	DS	
D PartNumber		INZ('13CHRYEXP12345A')
D ModelYear	2A	OVERLAY(PartNumber)
D Platform	4A	OVERLAY(PartNumber:*NEXT)
D Model	3A	OVERLAY(PartNumber:*NEXT)
D EngrNum	6A	OVERLAY(PartNumber:*NEXT)
D InternalPart	9A	OVERLAY(EngrNum:1)

Figure 7 – Complex D-specs example of OVERLAY keyword use

Figure 7 shows two different ways of defining subfield overlay positions, explicitly with (name:position) and implicitly with (name:*NEXT). Specifying OVERLAY(name:*NEXT) positions the subfield at the next available position within the overlaid field. (This will be the first byte past all other subfields prior to this subfield that overlay the same subfield.)

For more information on Definition specifications see the RPG ILE reference manual.

Use Modern Techniques to Handle Numbered Indicators

RPG free-format forces programmers to use modern coding techniques, since it doesn't support many of the older RPG features including numbered indicators. Programmers must learn to replace the old style operation codes with use of the modern techniques. For the most part, numbered indicators are no longer used, but instead replaced with far more meaningful functions and named indicators. BIFs – built in functions have all but eliminated the need to use numbered indicators. Yet there are still instances where a numbered indicator may be used. With the introduction of RPG IV, enhancements included the following:

- Built-In Functions (BIFs)
- The Indicator Data Structure (INDDS)
- Logical Expressions
- Named Indicators

As a result the only indicator left that must be used is the *INLR indicator.

Built In-Functions

We no longer need to use resulting indicators on operation codes. Instead, we can use the relevant I/O BIF. The I/O BIFs are %OPEN, %FOUND, %EOF, %ERROR, %EQUAL and %STATUS. The code below shows an example of using the %OPEN, %ERROR and %FOUND BIFs. The %OPEN BIF indicates whether or not the file is open. The %ERROR BIF is set when the E extender is defined on an operation code (Chain in this case). We use the E extender in place of an error indicator in the low position. The %FOUND BIF is set automatically by the Chain operation.


```

If %Open(FileB);
  Chain(E) Key FileB;
  Select;
  When %Error;
    Dsply 'File Error';
  When %Found(FileB);
    Dsply 'Got It';
EndIf;

```

Figure 8 – BIF example with %OPEN, %ERROR and %FOUND

It is easy to see use of the BIFs makes it quite easy for a programmer to understand the program logic. %FOUND is more meaningful rather than using *IN50 = *OFF.

Indicator Data Structure

By default, the indicators 01 to 99 on a display or print file are mapped to the indicators 01 to 99 in an RPG program. But when we use the file keyword INDDS (Indicator Data Structure), they are mapped to a data structure instead of the indicators. In other words, the 99 indicators used with the display or print file are now associated with the data structure instead of the indicators *IN01 to *IN99 in the program and can be given meaningful names instead of using numbered indicators.

Logical Expressions

Figure 9 shows the traditional means of setting an indicator; it should look familiar. It is setting an error condition if it is a month end and if the status is neither 'I' nor 'C'. At least I think it is setting an error condition; that is what indicator 31 is -- right?

```

If *In31 = *Off;
  If EndMonth = 'Y';
    If Status <> 'I' and Status <> 'C';
      *In31 = *On;
    Endif;
  Endif;
Endif;

```

Figure 9 – Non logical expression way of setting an indicator

Figure 10 shows the same piece of code as a logical expression. The result of a logical expression is true or false, therefore it can be assigned directly to an indicator. Also, theEndMonth field has been replaced by the indicator MonthEnd, and *IN31 has been replaced by the indicator BadStatus.

```

If MonthEnd;
  BadStatus = (Status <> 'I' and Status <> 'C');
EndIf;

```

Figure 10 – Logical expression example

Named Indicators

Indicators are now a recognized data type in RPG, so we can define them in the D Specs, as shown below. Note that when testing an indicator you do not need to compare it against *ON or *OFF.

```
D BadStatus      S      N
D MonthEnd       S      N
```

Figure 11 – Named indicators example

The following code shows a named indicator being used for printer overflow. This is a simple example of the benefit of a named indicator; the name Overflow is more self-explanatory than *IN96 or *INOV. Also, you don't need to define the Overflow indicator in the D Specs; the compiler will define it for you.

```
FReport O E Printer OfIInd(OverFlow)
```

```
Write Detail;
```

```
If OverFlow;
```

```
    Write Header;
```

```
    OverFlow = *Off;
```

```
EndIf;
```

Figure 12 – Named indicator example

We have reviewed a few examples of the modern techniques to handle numbered indicators. There are many more areas that have been impacted by modern techniques of RPG coding that eliminate the need to use numbered indicators. Today's programmer should be familiar with the language options and techniques.

For more information on named indicators and coding techniques refer to the RPG ILE reference manual.

Declare and Use Subprocedures

Subprocedures are a very useful tool that can be used within a program and reused throughout an application and system. Subprocedures can be used to create a library of routines that can be reused by any RPG or CL program on a system. They are isolated routines similar to standalone programs. A subprocedure is defined within a program, after the main source section. No RPG cycle code is generated for subprocedures; therefore, you cannot code pre run-time and compile-time arrays and tables, *DTAARA definitions or Total calculations. Within a subprocedure, declared fields are unique and independent of defined fields within a program. Fields defined within subprocedures are referred to as local variables. A subprocedure can make use of local and global variables. The content of the subprocedure fields exists while the subprocedure is running and no longer exist when the subprocedure ends. Each time a subprocedure is called the fields are initialized.

Subprocedures are used with prototyping, passing parameters and returning values. A prototype is a list of parameters used by a subprocedure and are defined using D-specs. Specifying a prototype within a module that defines the subprocedure is optional; however, it should not be considered optional when a subprocedure is exported from a module and the procedure is called from other RPG modules or CL programs. When exported, a prototype should be specified using a copy file and copied into a module that defines the subprocedure and into any module that calls the subprocedure.

Figure 13 is an example of a subprocedure that is used to pass a part number and user id and return a formatted message that includes the part number and user id within the message.

```
D PartMsg      PR      64A
D
D      25A
D      103A

D $part              25A
D $userid      S      10A
D $message      S      64A

/free

.....

$message = PartMsg($part:$userid);
dsply $message;

/end-free

P PartMsg      B      EXPORT

D PartMsg      PI      64A
D $part              25A
D $userid              10A

D $message      S      64A

/free

$message = 'Part ' + %trim($part) + ' was last modified ' +
           'by ' + %trim($userid) + ";
return $message;

/end-free

P PartMsg      E
```

Figure 13 – Example of subprocedure definition and use.

For additional information on subprocedures, see the RPG ILE reference manual.

Use Externally Described Data Structures

Data structures can be program described or externally described with the exception of indicator data structures, which are program described only. A data structure can be defined like another using the LIKEDS keyword. An externally described data structure will have an E in position 22 of the D-spec. The E in position 22 tells the compiler the data structure uses the external name to retrieve the description of the data structure subfields. The name of the external file can be placed in either positions 7 through 21 or as a parm for the D-spec keyword EXTNAME. An external subfield name can be renamed in the program using the EXTFLD keyword. The PREFIX keyword can be used to add a prefix to the external subfield names.

Declare an externally-described data structure based on the customer file named CUSTMAST.

```
D CustRec      E DS          ExtName(CUSTMAST:*KEY)
```

Figure 14 – Declaring an externally described data structure

The external description is used as the internal format of the subfields. The last parm specifies which fields in the external record to extract.

EXTNAME(file-name{:format-name}{:*ALL|*INPUT|*OUTPUT|*KEY})

- ***ALL** extracts all fields.
- ***INPUT** extracts just input capable fields.
- ***OUTPUT** extracts just output capable fields.
- ***KEY** extracts just key fields.

Programmers can specify search arguments in keyed Input/Output operations in free- format calculations in two ways, by specifying search arguments or by specifying a data structure which contains the search arguments.

/free

```
CHAIN (key1 : key2 : key3 : key4) CUSTMAST;  
CHAIN %KDS(CustRec) CUSTMAST;
```

Figure 15 – Specifying search arguments in keyed Input/Output operations

The INFDS keyword lets you define and name a data structure to contain the feedback information associated with the file. The data structure name is specified as the parameter for INFDS. If INFDS is specified for more than one file, each associated data structure must have a unique name. An INFDS can only be defined in the main source section.

The INFDS contains the following feedback information:

- File Feedback (length is 80)
- Open Feedback (length is 160)
- Input/Output Feedback (length is 126)
- Device Specific Feedback (length is variable)
- Get Attributes Feedback (length is variable)

For more information on externally described data structures see the ILE RPG Reference manual.

Write Logic (including I/O operations) Without Numbered Indicators

RPG Free forces programmers to use modern coding techniques, since it doesn't support many of the older RPG features including numbered indicators. Programmers must learn to replace the old style operation codes with use of the modern techniques. For the most part, numbered indicators are no longer used, but instead replaced with far more meaningful functions and named indicators. BIFs – built in functions have all but eliminated the need to use numbered indicators.

Resulting indicators on operation codes are no longer needed. Instead BIFs are used for file and record status. I/O BIFs including %EOF, %ERROR, %EQUAL, %FOUND, %OPEN and %STATUS replace use of a number to indicators. Instead of using a numbered indicator after a file operation and then testing the value of the indicator to recognize whether or not a record is retrieved, a BIF will be used as a condition test within your program logic. Elimination of the use of numbered indicators adds significant value replacing a number with a much more meaningful indicator. No longer will the HI, LO and EQ resulting indicators be used on a C Spec. No longer are you required to make sure to put the numbered indicator in the correct column instead a BIF can be used in free-format.

%EOF Return End or Beginning of File Condition	Operations that set %EOF include READ, READC, READE, READP, READPE and WRITE (subfile only) %EOF if successful is set to off by the following operations if the open is not successful; CHAIN, OPEN, SETGT and SETLL
%ERROR Return Error Condition	The operations that set %ERROR include; ACQ, ADDUR, ALLOC, CALL, CALLB, CALLP, CHAIN, CHECK, CHECKR, CLOSE, COMMIT, DEALLOC, DELETE, DSPLY, EXFMT, EXTRCT, FEOD, IN, NEXT OC-CUR, OPEN, OUT, POST, READ, READC, READE, READP, READPE, REALLOC, REL, RESET, ROLBK, SCAN, SETGT, SETLL, SUBDUR, TEST, UNLOCK, UPDATE, WRITE, XLATE
%EQUAL Return Exact Match Condition	The operations that set %EQUAL include SETLL and LOOKUP.
%FOUND Return Found Condition	The operations that set %FOUND are CHAIN, DELETE, SETGT, SETLL, string operations CHECK, CHECKR, SCAN and search operation LOOKUP.
%OPEN Return File Open Condition	Used to check if a file has been opened by the RPG module during initialization or by the OPEN operation.
%STATUS Return File or Program Status	%STATUS returns the most recent value set for a file or program status.

Figure 16 – I/O BIFs used to replace use of numbered indicators.

The following example reflects use of the %OPEN, %ERROR and %Found BIFs. Figure 17 reflects an example of modern code using I/O operations without numbered indicators.

```
If %Open(CustMaster);
  Chain(E) CustNumber CustMaster;
  Select;
  When %Error;
    Dsply 'Customer Error';
  When %Found(CustMaster);
    Dsply 'Customer Found';
EndIf;
```

Figure 17 – Example File I/O modern techniques.

Named indicators can be defined within D specs and used for testing. Figure 18 demonstrates how a named indicator may be defined and used.

```
D YearEnd      S      N

/free
  If YearEnd = *On ;
    Exsr YearEndRoutine;
  Endif;
/End-free
```

Figure 18 – Named indicator defined in D-spec example.

Another change in indicator usage can be seen by the Figure 19. Instead of using *INOF a named indicator 'PageOverFlow' is being used. The overflow indicator doesn't need to be defined within D-specs. By referencing the name indicator on the printer file specification the compiler defines the indicator for you.

```
FGLReport O  E      Printer OfIInd(OverFlow)

/free
  Write GLRptDetail;
  If OverFlow;
    Write GLRptHeader;
    OverFlow = *Off;
  EndIf;
/end-free
```

Figure 19 – Named indicator referenced on the printer file specification.

We have reviewed a few examples of the modern techniques to handle numbered indicators. There are many more areas that have been impacted by modern techniques of RPG coding that eliminate the need to use numbered indicators.

Today's programmers should be familiar with the language options and techniques. For more information on named indicators and coding techniques refer to the RPG ILE reference manual.

Coding and Using Free-Format Calculation Specifications

The introduction of free-format calculation specifications was a significant enhancement to the RPG language providing a broad range of functionality that significantly enhanced RPG. Bryan Meyers from Linoma wrote a guide reference for RPG programmers, which includes the 7 rules of RPG free-format. They are useful in understanding the rules in which a programmer must abide by to code using free format.

7 Rules of Free-Format RPG

1. Free-format RPG coding begins with a /FREE directive (in position 7) and ends with an /END-FREE (also in position 7). Between these directives, you code free form statement lines.
2. Free-format statements can begin anywhere in positions 8-80. Positions 6-7 must be blank. Indenting is allowed.
3. Free-format statements begin with an opcode (and extender, if any) followed by Factor 1, Factor 2, and the result field operands separated by blanks. Operands that are not required may be omitted.
4. Free-format statements end with a semicolon (;).
5. Comments begin with double slash characters (//) and can be placed anywhere on the line starting in position 8. A comment can be code on a line of its own or can follow "inline" after a free-format statement.
6. The portion of a line after the semicolon must be blank or contain an inline comment.
7. Level indicators, conditioning indicators, and resulting indicators are not allowed. Resulting indicators are replaced by built in functions.

Positions	Name	Entry
6-7		Blank
7	Start or end free-form specification	/FREE or /END-FREE delimiter
8-80	Free-form calculation statement	Operation code and extendor, expression ended with a semi-colon (;)
6-80	Comments	//

Figure 20 - Free-Format Calculation Specifications

Free-format RPG calculations are much different from traditional RPG calculations and often programmers with years of experience using fixed-format have a transition to master the coding technique. At first glance it isn't familiar and takes a little getting used to. Most schools that teach RPG today as a part of their curriculum use free-format. Some younger developers have an adjustment when asked to work on and use fixed format code finding requirement to have everything in its proper column, using indicators, shorter field names and not being able to indent code frustrating and confusing to understand. Mastering and using free-format coding techniques not only can make a developer more productive, but when used properly and documented can also make their code much easier to follow and understand. Like any coding

technique it is best learned by understanding the syntax and practicing. It's not hard to find examples of code on the internet and there are also many publications available to use as learning tools and reference.

Built in functions are one of the best features of the modern RPG language. I very much appreciate BIFs, as do many other RPG programmers. I can't imagine coding free-format RPG without making use of BIFs. BIFs are similar to operation codes and they can be used within expressions. Constant valued BIFs can be used in named constants. The name constants can be used within any specifications. There is a long list of BIFs. BIFs are easily recognized as they start with a %. The naming convention for BIFs is very logical in identifying the use of the BIF. For example %EOF used to determine end of file condition or %DAYS which returns the number of days as a duration. Earlier, in reference to writing logic without numbered indicators, we touched upon some of the BIFs used for file and record status including %EOF, %ERROR, %EQUAL, %FOUND, %OPEN and %STATUS. BIFs can be used for a variety of purposes for example %CHAR is used to convert a character field to a numeric field, %DIFF is used to return the difference between two dates, times or timestamps, %TRIM and %TRIMR are used to trim blanks from a field. Duration BIFs including %HOURS, %MINUTES, %MONTHS, %MSECONDS, %SECONDS are used to return an equivalent value as a duration. %DATE and %TIME are used to return equivalent system values.

For a complete list of BIFs see the RPG ILE reference manual.

/Free

```
// Convert from a char to Num
```

```
num = %DEC(char);
```

```
// Chain to the correct record, checking for error with no rec lock
```

```
CHAIN(EN) (char) filename1;
```

```
IF %FOUND(filename1) and NOT %ERROR; // If Condition
```

```
CHAIN (num) filename2; // Chain to next file
```

```
IF %FOUND(filename2); // If Condition
```

```
filevar1 += 1;
```

```
filevar2 = char;
```

```
filevar3 = proc_call(filevar1:filevar2); // Call a Procedure
```

```
// Update only field filevar3 in filename2
```

```
UPDATE filerec2 %FIELDS(filevar3);
```

```
ENDIF;
```

```
ELSEIF %ERROR;
```

```
DSPLY 'There was an error';
```

```
*INLR = %ERROR;
```

```
RETURN;
```

```
ENDIF;
```

```
*INLR = *ON;
```

```
RETURN;
```

/End-Free

Figure 21 – Example of RPG free-format calculations.

```
/free

read file;          // Get next record
dow not %eof(file);  // Keep looping while we have
                    // a record
if %error;
    dsply 'The read failed';
    leave;
else;
    chain(n) name database data;
    time = hours * num_employees
        + overtime_saved;
    pos = %scan (':', name);
    name = %xlate(upper:lower:name);
    exsr handle_record;
    read file;
endif;
enddo;

begsr handle_record;
    eval(h) time = time + total_hours_array (empno);
    temp_hours = total_hours - excess_hours;
    record_transaction();
endsr;

/end-free
```

Figure 22 – Free-format RPG calculation specification example.

For a complete list of Opcodes, BIFs and Compiler directives refer to RPG ILE Reference manual.

Translate Operation Codes not Supported in Free-Format (e.g., MOVE, CALL, etc.) into Free-Format

Whether you are new to RPG and have learned free-format RPG as your first introduction to RPG or you are an experienced RPG programmer with years of experience, when using older versions of RPG, it is important to be able to translate operation codes not supported in free-format. Using free-format is encouraged and is vital for a programmer in a modern day IBM i environment.

Most, but not all, of the older operation codes used within RPG are supported in RPG free-format. Many of the operation codes have been replaced by BIFs, but others don't necessarily have a direct equivalent. For example ADD, SUB, DIV, MULT have been replaced with use of +, -, /, *. The MOVE and MOVE* operation codes are not supported by RPG Free. Instead, BIFs including %CHAR, %DEC, %DATE, %SUBST and other built in functions will be used. At first glance, free-format may seem drastically different, but once a programmer is comfortable there are benefits to be had in readability, indentation, ability to complex calculations with fewer lines of code and more.

Converting the fixed-format MOVE related operation codes to free-format is dependent upon the data type and size of the source field and target fields. Figure 23 is a table that includes some of the possibilities.

Source Field	Target Field	Free Format Operations
Character	Character	Use EVAL,EVALR, with or without %SUBST.
Character	Numeric	Use %DEC, %INT, %UNS functions.
Character	Date	Use %DATE, %TIME, %TIMESTAMP functions.
Date	Character	Use %CHAR function.
Date	Numeric	Use %DEC, %INT, %UNS functions, with %CHAR function.
Date	Date	Use simple EVAL.
Numeric	Character	Use %EDITC function with X edit code, with or without %SUBST function.
Numeric	Numeric	Use simple EVAL.
Numeric	Date	Use %DATE, %TIME, %TIMESTAMP functions.

Figure 23 – Example of Free-Format operations used to replace MOVE operations.

Figure 24 is an example of a MOVE operation and an equivalent RPG Free example.

D Field1	S	30A	
D Field2	S	10A	
D Field3	S	35A	
C	MOVEL	Field1	FLD3
C	MOVE	Field2	FLD3
/free			
Field3 = %subst(Field1:1:25) + Field2;			
/end-free			

Figure 24 – example of MOVL and MOVE translated to free form

Figure 25 shows an example of arithmetic calculations.

L0N01N02N03Factor1+++OpcdeFactor2+++ResultLenDHHiLoEqComments++++++.....

```

C      TMSFT1  MULT EMRATE  TMPAY      Shift 1 pay
*
C      EMRATE  ADD  EMPRE2  XRATE      Shift2 rate + premium
C      TMSFT2  MULT XRATE   XWAGET     Shift2 wages
C              ADD  XWAGET  TMPAY      Shift1+2 pay

```

*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 .8+....9

```

CL0N01Factor1++++++Opcode(E)+Extended-factor2++++++Comments++++
C          eval    tmpay = tmsft1 * emrate          Shift1 wages
C          + (emrate + empre2) * tmsft2 Shift2 wages
*          including premium

/free

    tmpay = tmsht1 * emrate + ( (emrate + emre2) * tmsft2);

/end-free

```

Figure 25 – Example translate operation codes not supported in free-format.

There are many other calculations that translate differently including program calls.

Recently I learned from Barbara Morris from IBM Toronto labs that there is a feature in RDp that can be used to translate older versions of RPG code to free-format code. Although the translation may require a bit of cleanup, it can be used to assist a developer in the transition to using free-format RPG or may also be used for those familiar with RPG Free and not older versions of RPG to gain an understanding of the equivalents in RPG Free.

Use Qualified Data Structures (e.g., LIKERECD, LIKEDS)

The keyword QUALIFIED indicates that subfields of the data structure are referenced using qualified notation. This permits access by specifying the data structure name followed by a period and the subfield name, for example DS1.FLD1. If the QUALIFIED keyword is not used, the subfield name remains unqualified, for example FLD1. The subfields can have any valid name, even if the name has been used elsewhere within the program.

LIKERECD keyword is used to define a data structure, data structure subfield, prototype return value or prototyped parameter like a record. The subfields of the data structure will be identical to the fields in the record. LIKERECD can take an optional second parameter which indicates which fields of the record to include in the data structure by using include.

The format of the keyword is LIKERECD(intrecname{:*ALL|*INPUT|*OUTPUT|*KEY}).

- ***ALL** All fields in the external record are extracted.
- ***INPUT** All input-capable fields are extracted. (This is #the default.)
- ***OUTPUT** All output-capable fields are extracted.
- ***KEY** The key fields are extracted in the order that the keys are defined on the K specification in the DDS.

Even if a field in the record is explicitly renamed on an input specification, the external name (possibly prefixed) is used, not the internal name. A data structure defined with the LIKERECD is a QUALIFIED data structure. The names of the subfields will be qualified with the new data structure name for example DS1.SUBF1. LIKERECD can be coded for subfields of a qualified data structure. When LIKERECD is coded on a data structure subfield definition, the subfield data structure is automatically defined as QUALIFIED. The subfields defined with LIKERECD are themselves data structures and can be used wherever a data structure is required.

```

FCUSTMAST IF E    K DISK          PREFIX('CM.')
FCUSTMAST1 IF E    K DISK          PREFIX('LGL.')
F                                RENAME(CUSTREC : CUSTLGL)

D CM                DS            LikeRec(CUSTREC)
D LGL                DS            LikeRec(CUSTLGL)
D save                DS            LikeRec(CUSTREC)

```

Figure 26 - Example LIKERECD keyword use.

LIKEDS keyword is used to define a data structure, data structure subfield, prototyped return value or prototyped parameter like another data structure. The subfields of the new item will be identical to the subfields of the other data structure. The format of the keyword is LIKEDS(data_structure_name).e)

```
D sysName      DS      qualified
D lib          10A inz(*LIBL)
D obj          10A
D userSpace    DS      likeds(sysName)inz(*LIKEDS)

// The variable "userSpace" was initialized with *LIKEDS, so the
// first 'lib' subfield was initialized to '*LIBL'. The second
// 'obj' subfield must be set using a calculation.

/free
userSpace.obj = 'TEMPSPACE'
```

Figure 27 - Example of using inz(*LIKEDS).

The names of the subfields will be qualified with the new data structure name. An unqualified subfield named.subfield or qualified subfield named dsname.subfield will result in a new subfield named dsname.subfield. LIKEDS can be coded for subfields of a qualified data structure. When LIKEDS is coded on a data structure subfield definition, the subfield data structure is automatically defined as QUALIFIED. Subfields of a LIKEDS subfield data structure are reference in fully qualified form: "ds.subf.subfa". Subfields defined with LIKEDS are themselves data structures and can be used wherever a data structure is required. The values of ALIGN and ALTSEQ keywords are inherited by the new data structure. The values of keywords OCCURS, DIM, NOOPT and INZ are not inherited. To initialize subfields in the same way, parent data structure specify INZ(*LIKEDS).

Prototype program calls

A prototype is a definition of the program call interface and includes the following information:

- Whether the call is bound (procedure) or dynamic (program).
- How to find the program or procedure (the external name).
- The number and nature of the parameters.
- Which parameters must be passed, and which are optionally passed.
- Whether operational descriptors should be passed.
- The data type of the return value, if any (for a procedure).

When the procedure is called from a different RPG module the prototype must be explicitly specified within both the calling module and the module that defines the procedure. The procedure is explicitly defined if it is only called within the same module or it may be omitted. If it is omitted the compiler will implicitly define it from the procedure interface. A prototype is defined using the EXTPGM or EXTPROC keyword in positions 7 through 21. The CALLP operation is used to call prototyped procedures or programs. Unlike the other call operations, CALLP uses free-form syntax. You use the |name operand to specify the name of the prototype of the called program or procedure, as well as any parameters to be passed. Using a prototype replaces the need to use a *ENTRY PLIST instead using PR/PI definitions. The coding of a prototyped program is consistent with coding required for subprocedures. Note that if CALLP is used to call a procedure which returns a value, that value will not be available to the caller. If the value is required, call the prototyped procedure from within an expression.

- * This prototype for QCMDexc defines two parameters:
- * 1- a character field that may be shorter in length than expected
- * 2- any numeric field

```
D qcmdexc      PR          extpgm('QCMDexc')
D  cmd         200A  options(*varsize) const
D  cmdlen      15P 5 const

/FREE
  qcmdexc ('WRKSPLF': %size ('WRKSPLF'));
/END-FREE
```

Figure 28 – Calling a Prototyped Program Using CALLP.

CALLP is implicit in this example and can be omitted. Inserting the CALLP and explicitly coding the call will provide the same result.

```
/FREE
  CALLP qcmdexc ('WRKSPLF': %size ('WRKSPLF'));
/END-FREE
```

Figure 29 – Calling a Prototyped Program Using CALLP.

The operation code extender “E” can be specified to handle CALLP exceptions. To use Prototype calls requires an understanding of use of the code extender including E, M and R. An understanding of precision rules for numeric operations can help prevent exceptions and errors. The E extender is only active during the final call for CALLP. If an error occurs on a call that is done as part of the parameter processing, control will not pass to the next operation.

For more information on Prototype program calls see the RPG ILE Reference manual.

Determining the Appropriate Use of Passing Parameters by Value Vs. By Reference

RPG Program calls require that parameters are passed by reference. RPG ILE procedure calls can pass parameters either by reference or by value. Parameters that are not prototyped may only be passed by reference. Furthermore, if passed by reference, parameters can be passed by read only reference. When calling an existing program or procedure, parameters must be passed in the way the procedure expects.

Passing by Reference

Passing by reference is the default for ILE RPG. Parameters that are not prototyped and external program calls can only be passed by reference. If parameters are passed by reference, they do not require coding keywords. Parameters should be passed by reference to a procedure if it expected the calling module to modify the field(s) passed. Passing by reference may also improve run time performance and should be considered when passing large character fields.

Passing by Value

A prototyped procedure may also be passed by value instead of by reference. The compiler passes the actual value to the called procedure when a parameter is passed by value. The called program or procedure can change the value of a parameter when passing by value is used. The keyword VALUE is used on a parameter definition in a prototype to designate the parameter will be passed by value. Program calls cannot pass parameters by value.

Passing by Read-Only Reference

Passing by read-only reference may only be used by a prototyped procedure or program. Passing by read-only reference is useful when you know that the value of parameters will not be changed during a call. Passing by read-only has the same advantages as passing by value. This method allows you to pass literals and expressions. When passed by read-only reference, the compiler may copy the parameter to a temporary field and pass the address of the temporary field. This may be caused by passed parameter being an expression or having a different format.

Some of the advantages of passing by value or read-only reference include:

- Literals and expressions can be passed as parameters
- Allows passing parameters that do not match exactly the type and length that are expected
- Prevents passed variables from being modified
- Less stringent matching of attributes of passed parameter

Passing by value may have a slight performance impact. If a subprocedure is called once for each record in a large file you may want to instead pass parameters by reference.

See the RPG ILE reference manual for additional information.

Enumerate Advantages of Prototypes Compared with PARM and PLIST

RPG ILE provided capability to call a program or a procedure. Also in ILE, a program or procedure call can be prototyped or not prototyped. The recommended method to call a program or procedure is to use a prototyped call. The syntax used for calling and passing parameters to prototyped procedures or programs is the same free-form syntax that is used with expressions or BIFs. For this reason a prototyped call is sometimes referred to as a free-form call. The prototyped calls (CALLP or a function call) are just as efficient as CALL and CALLB and offer the advantages of prototyping and parameter passing by value. CALL and CALLB cannot accept a return value from a procedure. Only prototyped procedures can return values; prototyped programs cannot.

CALL or CALLB operations to call a program or procedure may be used when an extremely call interface is used, use of PARM operation with factor 1 and factor 2 is required or when more flexibility is required for parameter checking.

Some of the advantages of using Prototypes compared with PARM and PLIST include:

- Simpler to call
- Offers more functionality
- Allows calling programs that are on the system at run time
- Allows calling exported procedures in other modules or service programs that are bound in the same program or service program
- Allows calling subprocedures in the same module
- Variable length parameters can be used
- Free-form call syntax

To understand the advantages of using Prototypes in place of using PARM and PLIST a programmer needs to have an understanding of how to define prototypes and make use of the functionality of how a prototype can be used. This will require an understanding of the different types of prototypes and the keywords that can be used when defining a prototype.

For information on passing prototyped parameters, see [Passing Prototyped Parameters](#).

*Determine the Appropriate Use for Prototype Keywords Such as CONST, VALUE, and OPTIONS (*NOPASS,*OMIT, *VARSIZE)*

There are many prototype keywords that can be used to provide extended functionality when using prototypes. Some of the keywords available include CONST, VALUE and OPTIONS. An understanding of the keywords available to use, the syntax and purpose of the keyword is required to make full use of the functionality provided through use of prototypes. Some of the keywords available for use with prototypes include:

- ALTSEQ(*NONE)
- ASCEND
- CCSID(number | *DFT)
- CLASS(*JAVA:class-name)
- CONST{(constant)}
- DATFMT(format{separator})
- DESCEND
- DIM(numeric_constant)
- LIKE(name)
- LIKEDS(data_structure_name)
- LIKERECD(intreccname:{*ALL|*INPUT|*OUTPUT |*KEY})
- NOOPT
- OPTIONS(*NOPASS *OMIT *VARSIZE *STRING *TRIM *RIGHTADJ *NULLIND)
- PROCPTR
- TIMFMT(format{separator})
- VALUE
- VARYING

The VALUE keyword indicates that the parameter is passed by value rather than by reference. Parameters can be passed by value when the procedure they are associated with are called using a procedure call. The VALUE keyword cannot be specified for a parameter if its prototype was defined using the EXTPGM keyword. Calls to programs require that parameters be passed by reference.

The CONST keyword is used to specify the value of a named constant or to indicate that a parameter passed by reference is read-only. When specifying the value of a named constant, the value can be specified with or without the CONST keyword. The parameter must be a literal, figurative constant or built in function. If a constant is used as a parameter for the keywords DIM, OCCURS, PERRCD or OVERLAY, it must be defined prior to use. The CONST keyword on a prototype definition cannot be used unless the parameter will not be changed by the called program or procedure. A CONST parameter cannot be changed by statements within the procedure, but can be changed as a result of statements outside of the procedure or by referencing a global variable.

The OPTIONS keyword is used to specify one or more parameter passing options including *NOPASS, *OMIT, *VARSIZE, *STRING and *RIGHTADJ.

*NOPASS is specified on a definition specification when the parameter does not have to be passed on the call. *OMIT is specified, the value *OMIT is allowed for the parameter. *OMIT is only allowed for CONST parameters and parameters that are passed by reference. *VARSIZE is only valid for parameters passed by reference that have a character, graphic or UCS-2 data type or that represent an array. When *VARSIZE is specified, the passed parameter may be shorter or longer than defined in the prototype. The %LEN BIF can be used to determine the current length of a passed parameter. *STRING is specified for a basing pointer parameter passed by value or by constant-reference. Either a pointer or a character expression can be passed when using *STRING. *RIGHTADJ is specified for CONST or VALUE parameter in a function prototype. The character, graphic or UCS-2 parameter value is right adjusted. *RIGHT ADJ is not allowed for varying length parameters. Varying length parameters may be passed on a procedure call where corresponding parameter is defined with *RIGHTADJ. We have reviewed some of the keywords available for use with prototypes. To prepare for the exam all of the keywords available should be reviewed and understood. To learn more about prototype keywords refer to the IBM ILE reference manual.

Use EVAL-CORResponding

EVAL-CORR stands for Evaluate Corresponding. It is used to copy fields from one data structure to with the same name and compatible data definition in another data structure. To use EVAL-CORR, at least one of the data structures must be qualified to allow using subfields with the same name. Compatible data definition means allowing copy of a numeric to another numeric or character to another character field regardless of the length of the 2 fields. Unlike fields cannot be copied for example a character field cannot be copied to a numeric field and a numeric field cannot be copied to a timestamp. EVAL-CORR can be used in fixed-format or free-format RPG. When used in free-format H and R operation extenders can be used.

Figure 30 uses two physical files OLDFILE and NEWFILE.

* OLDFILE

```
A          UNIQUE
A  R DATAREC
A    FIELD1    5P 0
A    FIELD2    20A    ALWNULL
A    FIELD3     L    ALWNULL
A    FIELD4     1A
A    FIELD5     1A
A  K FIELD1
```

* NEWFILE

```
A          UNIQUE
A  R DATAREC
A    FIELD1    7P 0
A    FIELD2    20A
A    FIELD3     L    ALWNULL
A    FIELD4    1P 0    ALWNULL
A    FIELD5     1A    ALWNULL
A    FIELD6    3P 0    ALWNULL
A    FIELD7    5P 2    ALWNULL
A  K FIELD1
```

Figure 30 – Files used for EVAL-CORR definition.

The files have the following differences.

- Field1 has been increased in size from five to seven digits
- Field2 is no longer null-capable
- Field4 has been changed from character to packed decimal in type
- Field4 has become null-capable
- Field5 has become null-capable
- Field6 and Field7 have been added

Using the example files all of the data will be copied to the new file except for Field4.

```

D OldRec      E DS          ExtName(OLDFILE)
D                                     qualified
D NewRec      E DS          ExtName(NEWFILE)
D                                     qualified

```

```

/free
eval-corr NewRec = OldRec;

```

Figure 31 – EVAL-CORR Use Example.

The compiler will provide generated messages to let you know which fields will be copied and why others won't be copied. How much information received is dependent upon using OPTION keyword *XREF or *NOXREF when the program was compiled.

For more information on the EVAL-CORR operation see the RPG ILE Reference manual.

Debug RPG Programs

Debugging programs is something a programmer will surely need to make use of. Whether it be for troubleshooting and correcting an application or to understand the existing logic of a program, debug will be a useful tool. Information on how to use the source debugger can be found in the online information and in the programmer's guide for the RPG ILE language.

To have debug data available requires compiling using the DBGVIEW on the CRTBNDRPG or CRTRPGMOD command. The parameter entered on the DBGVIEW determines what type of data if any is available. The system value can be defaulted to provide a default DBVIEW of choice. The options include *NONE, *STMT, *SOURCE, *COPY, *LIST and *ALL.

You can add more programs to and remove programs from a debug session, after starting a debug session. You must have *CHANGE authority to a program to add it to or remove it from a debug session.

To make use of the source debugger requires your current session to be in debug mode. Debug mode is a special mode which makes program debug functions available in addition to normal system functions. A session is put into debug mode when the STRDBG, start debug, command is run and will stay in effect for the session until the session is ended or the ENDDBG, end debug, command is run. ILE programs make use of the ILE debug environment.

The options used to compile a program will determine whether you are able to display source in debug mode. Debug may be used to display a program variable value using the debug EVAL statement. Breakpoints can be added within a program to stop the program execution at a specific line within the program. STEP or F11 can be used to step through the code 1 line at a time. WATCH allows you to request a breakpoint when the contents of a specified storage location is changed from its' current value. EVAL may also be used to change the value of a field.

Exception Handling

A developer may also be required to debug a program as a result of exception handling.

The programmers Exception handling process includes:

- Examining an exception message which has been issued as a result of a run-time error
- Optionally modifying the exception to show that it has been received (that is, handled)
- Optionally recovering from the exception by passing the exception information to a piece of code to take any necessary actions.

When a run-time error occurs and exception message is generated. An exception has one of the following types depending on why the error occurred:

- ***ESCAPE** -Indicates that a severe error has been detected.
- ***STATUS** -Describes the status of work being done by a program.
- ***NOTIFY** -Describes a condition requiring corrective action or reply from the calling program.
- **Function Check** -Indicates that one of the three previous exceptions occurred and was not handled.

Control specification can be used to control behavior of a program when an error is encountered. The control specification is designated by an H in position 6. The DEBUG keyword determines whether DUMP operations are performed and whether unused externally described input fields are moved from the buffer during input operations. DUMP operations are performed if keyword DEBUG or DEBUG(*YES) are specified on the control specification. DEBUG(*NO) can be used by specifying operation extender A, so a dump is always performed regardless of the DEBUG keyword.

We've touched lightly upon Debugging RPG programs. In preparation for the exam review the programmer's guide for the RPG ILE language. The exam does not include use of the RDp debugger.

Diagnose and Eliminate Errors for Date Data Types and Arithmetic Overflow

Date fields have a predetermined size and format. They can be defined on the definition specification. Leading and trailing zeros are required for all date data. Date constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators. Also, dates used for I/O operations such as input fields, output fields or key fields are also converted (if required) to the necessary format for the operation.

The default internal format for date variables is *ISO. This default internal format can be overridden globally by the control specification keyword DATFMT and individually by the definition specification keyword DATFMT.

The hierarchy used when determining the internal date format and separator for a date field is

- From the DATFMT keyword specified on the definition specification
- From the DATFMT keyword specified on the control specification
- *ISO

There are three kinds of date data formats, depending on the range of years that can be represented. This leads to the possibility of a date overflow or underflow condition occurring when the result of an operation is a date outside the valid range for the target field. The formats and ranges are as follows:

Number of Digits in Year	Range of Years
2 (*YMD, *DMY, *MDY, *JUL)	1940 to 2039
3 (*CYMD, *CDMY, *CMDY)	1900 to 2899
4 (*ISO, *USA, *EUR, *JIS, *LONGJUL)	0001 to 9999

Figure 32 – Date data formats.

Understanding the different date data types, definition, compatibility, usability, initialization and values will help a programmer diagnose and eliminate errors for date data types.

Integer and Unsigned Arithmetic

For all arithmetic operations (not including those in expressions) if factor 1, factor 2, and the result field are defined with unsigned format, then the operation is performed using unsigned format. Similarly, if factor 1, factor 2, and the result field are defined as integer or unsigned format, then the operation is performed using integer format. If any field does not have an integer or unsigned format, then the operation is performed using the default format, packed-decimal.

The following points apply to integer and unsigned arithmetic operations only:

- All integer and unsigned operations are performed in 8-byte form.
- Integer and unsigned values may be used together in one operation. However, if factor 1, factor 2, or the result field is signed, then all unsigned values are converted to integer. If necessary, unsigned 2-byte values are converted to 4-byte integer values to lessen the chance of numeric overflow.
- If a literal has 10 digits or less with zero decimal positions, and falls within the range allowed for integer and unsigned fields, then it is loaded in integer or unsigned format, depending on whether it is a negative or positive value respectively.

Integer or unsigned arithmetic may give better performance. However, the chances of numeric overflow may be greater when using either type of arithmetic.

The result field must be large enough to accommodate the results of the arithmetic operation or it may cause an arithmetic overflow error. If the result field is not large enough to accommodate the results, digits are dropped from either or both ends, depending on the location of the decimal point.

Code and use Monitor and %Error to handle runtime errors

ILE RPG provides four ways to enable HLL-specific handlers and to recover from an exception.

- error indicators or 'E' operation code extender
- MONITOR group
- INFSR error subroutine
- *PSSR error subroutine.

More information can be obtained about an error by coding appropriate data structures and querying relevant data structure fields. Using the 'E' extender instead of error indicators relevant program and file error information can be obtained by using the %STATUS and %ERROR BIFs. When an exception occurs within a main procedure ILE RPG does the following:

1. If an error indicator is present on the calculation specification and the exception is one that is expected for that operation:
 - a. The indicator is set on
 - b. The exception is handled
 - c. Control resumes with the next ILE RPG operation.
2. If an 'E' operation code extender is present on the calculation specification and the exception is one that is expected for that operation:
 - a. The return values for the built-in functions %STATUS and %ERROR are set.
 - b. The exception is handled
 - c. Control resumes with the next ILE RPG operation.
3. If no error indicator or 'E' extender is present and the code that generates the exception is in the MONITOR block of a MONITOR group, control will pass to the on-error section of the MONITOR group.
4. If no error indicator or 'E' extender is present, no active MONITOR group could handle the exception, and
 - you have coded a *PSSR error subroutine and the exception is a program exception
 - or
 - you have coded a INFSR error subroutine for the file and the exception is an I/O exception, the exception will be handled and control will resume at the first statement of the error subroutine.
5. If no error indicator, 'E' extender, or error subroutine is coded and no active MONITOR group could handle the exception, then the RPG default error handler is invoked.
 - If the exception is not a function check, then the exception will be percolated.
 - If the exception is a function check, then an inquiry message will be displayed. If the 'G' or 'R' option is chosen,

the function check will be handled and control will resume at the appropriate point (*GETIN for 'G' or the same calculation specification that received the exception for 'R') in the procedure. Otherwise, the function check will be percolated and the procedure will be abnormally terminated.

%ERROR can be used to avoid hard errors in an RPG program. To make use of %ERROR BIF the (E) option can be used on several commands for example READ(E), SETLL(E), UPDATE(E). When the %ERROR BIF is placed immediately after any of these commands it can be used to detect an error.

```
Read(E) CustMast;  
  If %Error = 1;  
    ErrorMessage = 'CustMast Read Error';  
  Endif'
```

Figure 33 – Example of %Error Code

%ERROR can be very useful on Read, Write and Update commands where there is a chance a file may be locked. Using operation code 'E' extender to capture the error and %ERROR BIF allows a programmer to control the program response when an error is encountered.

The MONITOR group performs conditional error handling based on a returned status code. It consists of using a MONITOR statement, one or more ON-ERROR groups and an ENDMON statement.

- * The MONITOR block consists of the READ statement and the IF
- * group.
- * The first ON-ERROR block handles status 1211 which
- * is issued for the READ operation if the file is not open.
- * The second ON-ERROR block handles all other file errors.
- * The third ON-ERROR block handles the string-operation status
- * The fourth ON-ERROR block (which could have had a factor 2
- * of *ALL) handles errors not handled by the specific ON-ERROR
- * operations.
- *
- * If no error occurs in the MONITOR block, control passes from the
- * ENDIF to the ENDMON.

```
Monitor;  
  Read FILE1;  
  If not %EOF;  
    Field1 = '1';  
  Endif;  
On-Error 1211;  
  ... handle file-not-open  
On-Error *FILE;  
  ... handle other file errors  
On-Error;  
  ... handle all other errors  
Endmon;
```

Figure 34 – Example of MONITOR use.

Using monitor captures potential error and allows a programmer some control on how the program responds to an error.

The ILE RPG Reference provides more information on how to handle runtime errors. See [Unhandled Exceptions](#) for a full description of the RPG default handler.

Additional Reference Materials

To prepare for the certification exam additional reference material should be reviewed including reference manuals, articles, books, blogs and other publications. Following are some useful links related to Advanced RPG & Problem Determination/Resolution.

- **DeveloperWorks – RPG Café**
<https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUid=b542d3ac-0785-4b6f-8e53-f72051460822>
- **IBMSystems Magazine – IBM i – Developer - RPG**
<http://www.ibmsystemsmag.com/ibmi/developer/rpg/>
- **ILE RPG Language Reference**
<http://publib.boulder.ibm.com/infocenter/iserics/v7r1m0/index.jsp?topic=%2Frzahg%2Frzahgrpglangref.htm>
- **iPro Developer (formerly System iNetwork)**
<http://www.iprodeveloper.com/>
- **IT Jungle – The Four Hundred Guru**
<http://www.itjungle.com/fhg/fhgindex.html>
- **MC-Press Online – Programming - RPG**
<http://www.mcpressonline.com/programming/rpg/>
- **Scott Klement's Webpage**
<http://www.scottklement.com/>
- **Midrange News**
<http://www.midrangenews.com/>
- **RPG Reference**
http://rpgreference.com/index.php/Main_Page
- **Search 400**
<http://search400.techtarget.com/>
- **Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More**
<http://www.redbooks.ibm.com/abstracts/sg245402.html?Open>

Some Noted Authors of RPG Articles and RPG Focused Books

Jim Buck, Charles Guarino, Robert Cozzi, Susan Gantner, Ted Holt, Scott Klement, Brian May, Barbara Morris, Brian Myers, Jon Paris and Paul Tuohy

ILE RPG Certification Prep - Part 3

by Brian May

In today's competitive marketplace, employers want to hire RPG developers with modern skills. If you are looking to land that great job, you need up-to-date skills. Of course, having the skills is only part of the solution. How do you prove to that potential employer or consulting customer that you have what they are looking for? One way is earn the COMMON Certified Application Developer – ILE RPG certification. This certification signifies that you not only possess a working knowledge of classic RPG programming, but you also have modern RPG and ILE skills.

The COMMON Certified Application Developer – ILE RPG exam is broken up into 4 sections to test competency in each area.

1. Core RPG, Subfiles, and Externally Described Files

This section of the exam will cover all of the basics that every RPG developer should know. Questions will cover using Externally Described Database, display and printer files, commonly used RPG opcodes, built in functions, subfile programming, and other core topics.

2. Advanced RPG and Problem Determination/Resolution

This section concentrates on more advanced RPG programming. Expect questions on data structure arrays, using named indicators, /free format coding, passing parameters by value/reference, debugging, and error handling.

3. RPG Data Handling

This section is the focus of this article. Topics covered include embedded SQL, using data structures for file I/O opcodes, converting to/from date data types, and string manipulation.

4. ILE

The final section is not necessarily about RPG coding. In this section your knowledge of the intricacies of ILE will be tested. Questions will cover procedure calls, activation groups, binding directories, service programs, and other related topics.

String Manipulation

If you have been writing business applications for very long, you have needed to manipulate character based data at some point. Maybe you needed to format data for output on a report or needed to parse incoming text data to be stored in a DB2 table. The CAT, CHECK, CHECKR, SCAN, SUBST, and XLATE opcodes have been around for a while, but our focus is modern RPG. The preferred method to perform string manipulation in modern RPG, whether fixed or /free format, is to use built in functions. There are several functions available.

- Concatenation

Originally in RPG, combining two strings required the use of the CAT opcode. In RPG IV, you can simply use the + operator. It makes perfect sense if you think of it as adding two strings together. If using no varying length character fields, remember that trailing blanks will still be included when concatenating.

- Trimming

Often useful when concatenating, the %TRIM, %TRIML, and %TRIMR functions allow you to remove leading or trailing blanks, or optionally other characters, from a character field used in an expression. The parameters for all three functions are the same. The first parameter is required and should be the character field to be trimmed. The second parameter is optional and tells the function what characters to trim. If the second parameter is not passed, blanks will be removed.

```
d Result          s          25a  Varying
d Example1        s          15a  Inz('      zzz  ')
d Example2        s          30a  Inz('aaaaa*****  ')

/free

Result = %Trim(Example1) ;
// Result now contains 'zzz'

Result = %TrimR(Example2 : '*' ) ;
// Result now contains 'aaaaa' ;

/end-free
```

- Modifying

There are multiple functions available for manipulating the contents of a character field. The most common of these is %SUBST. When used on the right hand side of the equal sign (=), %SUBST will return the desired segment of the character string. Most of us have done this at some point. Did you know that you can use %SUBST on the left side of an expression as well? When used in this manner, the value on the right side of the expression will replace the contents of only the segment matching the %SUBST selection. This is extremely handy.

```
d Result          s          25a  Varying
d Example1        s          15a  Inz('abcdefghijklm')
d Example2        s          30a  Inz('RPG can substring')

/free

Result = %Subst(Example1 : 4 : 3) ;
// Result now contains 'def'

%Subst(Example2 : 5 : 3) = 'did' ;
// Example2 now contains 'RPG did substring' ;

/end-free
```

Another common manipulation function is %XLATE. This function allows you to translate characters in a string to other characters. %XLATE requires three parameters. The first parameter is the list of individual characters to be replaced. The second is the list of characters used instead of the original characters. The third is the character field to translate.

```
d Result          s          25a  Varying
d UPCASE          c          'ABCDEFGHJKLMNOPQRSTUVWXYZ'
d LOWCASE         c          'abcdefghijklmnopqrstuvwxyz'

/free

Result = %Xlate(LOWCASE : UPCASE : 'I am mixed case') ;
// Result now contains 'I AM MIXED CASE'

/end-free
```

While %XLATE is great for replacing single characters in a one to one relationship, sometimes you need something a little more flexible. %REPLACE allows you to replace any number of characters with any number of new characters. With the proper parameter values, it can even insert or remove characters in the middle of a string.

The first parameter for %REPLACE is the string to be inserted. The second parameter is the string you are manipulating. The optional third and fourth parameters set the starting position and number of characters to replace. If not specified, the

starting position defaults to 1 and the number of characters to replace is set to the length of the first parameter.

```
d Result          s          25a  Varying
/free
Result = 'Brian is a great programmer' ;
Result = %Replace('terrible' : Result : 12 : 5) ;
// Result is now 'Brian is a terrible programmer'
Result = %Replace(' ' : Result : 12 : 9) ;
// Result is now 'Brian is a programmer'
/end-free
```

Often, you need to search a string to find where to insert or replace characters. The %SCAN function returns the position of the search argument in a string. The first parameter of %SCAN specifies the string you are looking for. The second is the character field you wish to search. The optional third parameter allows you to start search at a position other than the beginning of the field.

```
d Result          s          10i 0
d Text            s          35a
/free
Text = 'Brian is a great programmer' ;
Result = %Scan('great' : Text) ;
// Result is now 12
/end-free
```

In IBM i 7.1, a new function was added to make finding and replacing strings even easier. %SCANRPL scans through a character field and replaces all occurrences of a string with another string. Think of it as condensing a DOW loop, %SCAN, and %REPLACE down into a single function. The first three parameters are mandatory and are used to specify the string to search for, the replacement string, and the character field to search. The optional fourth and fifth parameters allow you to set the position to start the search and how many characters to search.

```
d Result          s          10i 0
d Text            s          100a
/free
Text = 'NAME is a great programmer. NAME said so.' ;
Result = %ScanRpl('NAME' : 'Brian' : Text) ;
// Result is now 'Brian is a great programmer. Brian said so.'
/end-free
```

Date Data Type

Many, if not most, DB2 tables used in RPG contain dates and times that are stored in columns defined as data types other than the native date, time, or timestamp types. This is often due to the age of the database as these types were not always available. Unfortunately, this practice is still sometimes used when defining new tables out of habit. Regardless of how the data ended up that way, eventually you will need to convert date data both to and from other data types. Luckily, RPG has handy built in functions to handle this.

- Converting to Date
Converting a date stored in a character or numeric field is very simple using the %DATE function. This function accepts

the character/numeric field containing the date data as the first parameter and the format of the date in the second. Time and Timestamp conversions are done using %TIME and %TIMESTAMP. These functions have the same parameters as %DATE.

```
d ISO_Num_Date      s          8s 0 Inz(20121225)
d YMD_Char_Date     s          8a  Inz('12/12/25')
d ISO_Char_Timestamp...
d                   s          30a  Inz('1960-09-29-12.34.56.000000')
d Result_Date       s          d
d Result_Timestamp...
d                   s          z

/free

Result_Date = %Date(ISO_Num_Date : *ISO) ;
// Result_Date is now d'2012-12-25'

Result_Date = %Date(YMD_Char_Date : *YMD) ;
// Result_Date is now d'2012-12-25'

Result_Timestamp = %Date(ISO_Char_Timestamp : *ISO) ;
// Result_Date is now 1960-09-29-12.34.56.000000

/end-free
```

- Date to Character

%CHAR allows you to convert a date, time, or timestamp to a character field. The first parameter is the date field. The optional second parameter specifies the format of the data returned.

```
d Date              s          d  Inz(d'2013-02-14')
d Character          s          10a

/free

Character = %Char(Date) ;
// Character is now '2013-02-14'

Character = %Char(Date : *USA) ;
// Character is now '02/14/2013'

/end-free
```

- Date to Number

%DEC functions identically to %CHAR but outputs a packed decimal instead of a character field

```
d Date              s          d  Inz(d'2013-02-14')
d Number            s          8p 0

/free

Number = %Dec(Date) ;
// Character is now 20130214

Number = %Dec(Date : *USA) ;
// Character is now 2142013

/end-free
```

Keys for I/O

Historically, retrieving records using RPG's record level access opcodes used KLISTs for record selection. The KLIST still exists and is still a valid option in modern RPG. But when coding in /free format, there is no way to define a KLIST without exiting back to fixed format. Luckily KLIST is no longer the only option for record selection on your favorite I/O opcode.

If you only need the selection of one or two I/O statements, it may be easier to specify your key directly on your I/O opcode. A list of key fields in parentheses and separated by commas can be directly on your I/O operation.

```

fMyFile      if      e              k Disk

d Key1              s              Like(MyKey1)
d Key2              s              3a

/free

Chain (Key1 : Key2 : '3') MyFile ;
If %Found(MyFile) ;
  //Do stuff
EndIf;

/end-free

```

If the key will be used multiple times, you will probably want a key list of some sort defined. You can certainly use the old KLIST opcode in fixed format, or you can use its replacement, a key data structure. Technically any data structure can be used as a key as long as the data types of the subfields match the file's key. So manually defining a data structure to use is an option, but IBM provided an easier way. You can define an externally defined data structure that contains only the key fields for the file. Once you have your data structure, you can use it with the %KDS function to treat it as a key.

```

fMyFile      if      e              k Disk

d MyKeyDS              ds              LikeRec(MyFileRec : *Key)

/free

Chain %KDS(MyKeyDS : 3) MyFile ;
If %Found(MyFile) ;
  //Do stuff
EndIf;

/end-free

```

As you can see, you can specify a second parameter to indicate how many of the key fields to use. This can reduce the number of keys needed. Unlike KLISTs, which required a new KLIST for each number of key fields used, you can have one key data structure per file no matter how many key fields each I/O operation requires.

Another really great feature of %KDS is that only the types of key fields need to match the file keys. If they are different lengths, or even zoned vs. packed decimal, %KDS will automatically convert the keys when used. No more temporary key fields to handle different lengths or different numeric data types. That will save you time and confusion.

Data Structures for I/O

Using data structures to send and receive file I/O is actually very simple. All you need to do is define your data structures using the EXTNAME keywords. Once you have the data structures they are added to the I/O statement in the result position.

```

fPOHeader1 uf a e              k Disk

d POHeader_In      ds              LikeRec(POHEADERR : *INPUT)
d POHeader_Out      ds              LikeRec(POHEADERR : *OUTPUT)

/free

Chain(n) (PO_Num) POHeader1 POHeader_In ;
If %Found(POHeader1) ;
  // Do Stuff
EndIf ;

Set11 (Start_PO) POHeader1 ;
Read(n) POHeader1 POHeader_In ;

Dow Not %EOF(POHeader1) ;
  //Do Stuff
  Read(n) POHeader1 POHeader_In ;
EndDo ;

Update POHEADERR POHeader_Out ;

/end-free

```

There are a few things to remember when using data structures for file I/O. In the example above, you will notice that we have data structures defined using *INPUT and *OUTPUT options. This is required by RPG. Basically your input only opcodes (CHAIN, READ, READE, READP, READPE, READC) are required to use a data structure defined using the *INPUT format. The WRITE opcode can only be used with the *OUTPUT format, while the UPDATE opcode can use either.

Using data structures for file operations has some advantages to consider. When data is retrieved from a file, the entire record is moved into the data structure in one large chunk instead of field by field. This, of course, is more efficient but also has the benefit of not causing the program to error out immediately if you have invalid data in a field. You can, for example, open a MONITOR block before first using a numeric field to trap a possible decimal data error and handle it. Also, if data structures are used for all I/O operations on a file, no I specs are generated by the compiler for that file. The elimination of I specs allows the use of some of the newer features of RPG such as using the ALIAS keyword when defining your data structure in order to use the longer SQL names for columns in the file. Data structures are also required if you are defining a file locally inside a subprocedure.

Embedded SQL

Obviously, teaching how to write SQL statements is beyond the scope of this particular article. But as there will be questions on the exam about using embedded SQL on the certification exam, you should know the basics of doing so. Embedded SQL allows SQL statements to be used directly within your RPG program to access database tables. Doing so requires the use of the /EXEC SQL directive in both fixed and /free format. After the directive, most SQL statements can be run.

```
C/EXEC SQL      UPDATE POHEADER SET STATUS = 'C'
C+              WHERE OPEN_QTY <= 0
C/END-EXEC

/free

  Exec SQL      DELETE FROM POHEADER
                WHERE STATUS = 'X';

/end-free
```

In most cases you will want to use variables from your RPG program in your SQL statement. This is easily accomplished by simply using a colon before the variable name in the SQL statement.

```
D Closed_Stat    S              1a  Inz('C')
D Cancel_Stat    S              1a  Inz('X')

C/EXEC SQL      UPDATE POHEADER SET STATUS = :Closed_Stat
C+              WHERE OPEN_QTY <= 0
C/END-EXEC

/free

  Exec SQL      DELETE FROM POHEADER
                WHERE STATUS = :Cancel_Stat ;

/end-free
```

If you need to select multiple records from a table and process them individually, you will need to use an SQL Cursor. Below is a very basic example of a cursor being processed in a loop.

```
/free

Exec SQL  DECLARE C1 CURSOR FOR
          SELECT PONUM, CUSTNO FROM POHEADER
          WHERE STATUS = 'O' ;

Exec SQL  OPEN C1;

Dow SQLCod = 0 ;

  Exec SQL FETCH C1 INTO :PO_Number, :Customer;
  // Do Stuff

EndDo;

Exec SQL  CLOSE C1;

/end-free
```

You should also do some reading on using prepared statements and parameter markers, otherwise known as Dynamic SQL. In its simplest form, you are marking the positions for variables using a question mark in the SQL statement and then providing the variables to be used at execution time.

```
D Statement      S          100a  INZ('SELECT PONUM, CUSTNO FROM    +
D                                     POHEADER WHERE STATUS = ?')
D Select_Status  S          1a
D PO_Number      S          8s 0
D Customer       S          10i 0

/free

Exec SQL  PREPARE S1 FROM :Statement ;

Exec SQL  DECLARE C1 CURSOR FOR S1 ;

Exec SQL  OPEN C1 USING :Select_Status ;

Dow SQLCod = 0 ;

    Exec SQL FETCH C1 INTO :PO_Number, :Customer;
    // Do Stuff

EndDo;

Exec SQL CLOSE C1;

/end-free
```

Embedded SQL is a huge topic to cover, so I recommend some independent study on the topic as we have only scratched the surface. Chapter 23 of the ILE RPG Language Reference is a good place to start (<http://publib.boulder.ibm.com/infocenter/iseriis/v7r1m0/topic/rzasd/sc0925081045.htm#sqlsyn>).

Additional Links

You should spend some time reading up on the topics covered in this article. Here are some links to get you started.

- ILE RPG Language Reference
<http://publib.boulder.ibm.com/infocenter/iseriis/v7r1m0/index.jsp?topic=%2Frzahg%2Frzahgrpglangref.htm>
- Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More
<http://www.redbooks.ibm.com/abstracts/sg245402.html?Open>
- Modernizing IBM eServer iSeries Application Data Access - A Roadmap Cornerstone
<http://www.redbooks.ibm.com/abstracts/sg246393.html?Open>
- DeveloperWorks – RPG Café
<https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=b542d3ac-0785-4b6f-8e53-f72051460822>
- IBMSystems Magazine – IBM i – Developer - RPG
<http://www.ibmsystemsmag.com/ibmi/developer/rpg/>
- IT Jungle – The Four Hundred Guru
<http://www.itjungle.com/fhg/fhgindex.html>
- MC-Press Online – Programming - RPG
<http://www.mcpressonline.com/programming/rpg/>
- iPro Developer (formerly System iNetwork)
<http://www.iprodeveloper.com/>
- Scott Klement's Webpage
<http://www.scottklement.com/>

About the Author:



Brian May is an IBM i modernization specialist for Profound Logic Software, as well as an industry writer, speaker, and educator. Brian began his career in 2001 as a software developer for a midsized apparel company. He serves on multiple COMMON committees, participates on industry advisory panels, and is a leader of the Young i Professionals (youngiprofessionals.com). As a writer and speaker, Brian specializes in topics related to RPG, PHP, open-source software, web technologies, and application modernization on IBM i. Email him at: brian@profoundlogic.com

ILE RPG Certification Prep - Part 4

by Birgitta Hauser

RPG Certification Prep Part 4 includes a lot of features and techniques to be accepted as a powerful programming language. But to exploit the full power of the programming language the ILE (Integrated Language Environment) concepts must be included. ILE does not only allow executing procedures, functions and methods written in other programming languages, but also provides coding techniques for highly modularized application development.

Before explaining the ILE concepts, let's start with the traditional (RPG) programming style.

OPM (Original Program Model) Main-Procedures

The Original Program Model (OPM) characterizes the traditional way to write (RPG) programs. Affected by the RPG cycle which originally was implemented to read a file from the beginning to the end and based on the procedural steps to be executed, those programs are written from top down. The RPG cycle is integrated in all RPG programs written in OPM style, independent whether a file is defined and read as IP (Input Primary) or UP (Update Primary) or not.

The complete source code for those programs is stored within a single source member, which results in monolithic and sometimes really enormous programs that are hard to maintain. To make the source code more manageable, huge programs are often split into multiple (sub-) programs to be called dynamically with the RPG Operation Code CALL. Dynamic Program call means, the program object is not resolved before it is called.

Even though all source code for a program is located within a single member, it can be structured by using subroutines. Subroutines allow the source code to be partitioned into smaller units that are coded only once but can be executed multiple times. With subroutines the source code becomes more readable. The downside however is subroutines are not encapsulated and only global variables defined within the global F, D or C-Specs can be accessed.

If a program must be modified the programmer has to have the general view over the program code.

Main-Procedures

All RPG statements before the first P-Statement, i.e. before the first (internal) procedure, are considered as (cycle) Main-Procedure. A (cycle) Main-Procedure can include H-, F-, D-, I-, C- and O-Specs. In this way all traditionally written RPG programs consist solely of a (cycle) Main-Procedure.

Beginning with Release 6.1, it is possible to also define (linear) Main-Procedures that do not include the RPG cycle. Linear Main-Procedures are coded like any exported procedure without return value, but the keyword EXTPGM has to be specified in the related prototype. To prevent the compiler from trying to integrate the RPG cycle the keyword MAIN (linearMainProcedureName) must be specified within the H-Specs.

Main-Procedures represent the program entry point, to be exact only Main-Procedures (linear or cycle) can be called directly from a command line a menu or can be submitted.

Sub-Procedures

Sub-Procedures often also referenced as internal procedures can be considered as improved subroutines, since they offer a number of significant advantages over subroutines:

- **Definition and Use of local Variables**

Within sub-procedures local variables that are only accessible from within the sub-procedure can be defined. Local variables with the same name and even with a different definition can be specified within multiple sub-procedures located within the same source member. These local variables are automatic, i.e.

they are newly created and initialized each time the sub-procedure is called. If a local variable has to keep its value between different calls, the keyword `STATIC` must be specified.

Even though it is also possible to access and modify global variables from within a sub-procedure, the use of global variables should be reduced to a minimum. To facilitate future modifications within the sub-procedure and/or program it is far better and easier to understand passing the global variable values (if necessary) as parameter values to the called sub-procedures.

- **Parameter passing and parameter checking**

Contrary to subroutines parameters can be defined and passed to sub-procedures.

Sub-Procedures require Prototyping for defining parameters and allow parameter checking at compile time. Before release 7.1 both, prototype and procedure interface, had to be specified for an internal procedure. Beginning with release 7.1 prototypes for internal procedures are optional, i.e. parameter checking is performed based on the procedure interface definition solely.

Prototyping allows parameters to be passed as input only parameter, either by value (Keyword `VALUE`) or constant reference (Keyword `CONST`). Additionally optional and/or omissible parameters can be defined with the keyword `OPTIONS(*NoPass)` or `OPTIONS(*OMIT)`. Besides `*OMIT` and `*NOPASS` all other parameters of the `OPTIONS` keyword can be used for sub-procedures, such as `*TRIM` (remove leading/trailing `*BLANKS`) or `*RIGHTADJ` (right adjust the input only parameter value).

- **Sub-Procedures can be defined as Function**

A function is a special type of a (sub-)procedure, more precisely a function is a procedure with return value. A function is not called through `CALLP` operation code (which is optional in RPG free format), but can be used like any RPG built-in-function, for example on the right side of the Equal operator (`=`) in an `EVAL` statement or in the extended factor 2 of an `IF`, `DOU` or `DOW` statement or as input only parameter in a procedure call.

The return value is defined within the `PR/PI` row of the prototype and procedure interface, simply by specifying the data type and length or using one of the keywords `LIKE`, `LIKEDS` or `LIKEREC`.

To return the return value to the caller it must be specified in factor 2 of the `RETURN` operation code.

- **Recursion**

The RPG cycle is only integrated in (cycle) Main-Procedures, but not in (sub-)procedures. The RPG cycle prevents RPG programs from being called recursively. Without the integrated cycle and because of the automatic local variables scope, (sub-)procedures can be called recursively. This can be useful in certain types of design, for example when analyzing BOMs (Bill Of Materials).

Ideally, (sub-)procedures should be coded as being “mini” programs or small black boxes, encapsulated, exchanging all information via parameter, using almost only local variables and local file access. Local F-Specs are allowed since release 6.1 and accessing data with embedded SQL was always local.

Exported Procedures and Functions

Exported procedures and functions are coded in the same way as internal procedures. There is only a single difference, the `EXPORT` keyword must be specified within the beginning P-Specification. Procedures or functions defined with the `EXPORT` keyword can be called from any other procedure located inside or outside the source member containing the source code for the procedure.

Contrary to main-procedures, exported procedures can only be called from other (internal or exported) procedures but cannot be called directly from a command line or a menu.

Since exported procedures and functions can be called from outside, prototypes are mandatory. The prototype of the exported procedure must be included in each source member from within the specific procedure that is called. To avoid hard-coding the same prototype over and over again, prototypes for exported procedures should be externalized into separate source members and integrated as copy members by using either the `/COPY` or `/INCLUDE` compiler directive.

Multiple exported procedures can be grouped together within the same source member. Since each exported procedure can be called from outside, each procedure has its own entry point, that means including a main procedure in the source member is not mandatory. To prevent the compiler from trying to include the RPG cycle in a module with internal and exported procedures only, the `NOMAIN` keyword must be specified within the H-Specs.

Anatomy of an (internal or exported) Procedure or Function

- **P-Specification: Begin and End of a Procedure**

A procedure always begins and ends with a P-specification. Within the beginning P-specification the procedure name must be specified.

The keyword EXTPROC is only necessary if the real procedure name differs from the procedure name to be used when calling the procedures or functions. The keyword EXTPROC should be used if your procedure is called from CL programs or procedures where the data type of the return value is handled differently by CL, for example the indicator data type (N).

If the keyword EXPORT is specified within the beginning P-statement, the procedure or function can be called from outside.

- **F-Specifications: Local File Access**

Local files can only be full-procedural files. Since the RPG cycle is not integrated, Input and Output specifications are not supported for procedures, so all input and output operations must be done using data structures for local files. The data structures can be defined as qualified or unqualified external data structures using the EXTNAME keyword or the LIKERECD keyword.

- **D-Specifications: Procedure Interface and Local Variables**

The procedure interface for the procedure/function has to be specified within the local D-specs.

All standalone variables, arrays, data structures and constants to be used solely within this procedure must be defined within the local D-Specs. Local variables are automatic and are initialized each time the procedure is called, except when the STATIC keyword is specified.

- **C-Specifications**

Local C-Specs include all RPG statements to be executed within the procedure.

A procedure is left as soon as a RETURN statement is performed. Ending the procedure with a RETURN statement is only required in functions for returning the return value to the caller procedure.

If subroutines are necessary for better structuring the procedure's source code, it is possible to include subroutines that can only be executed from within the procedure. But the more you learn to think modular the more you'll discover subroutines are not really necessary.

The following code snippet shows the skeletons of a procedure and an internal function. The procedure skeleton includes local F-Specs and subroutines, while the function code does not (even though it is possible to use F-Specs and Subroutines in functions). The only real difference between the procedure and function's code is the return value defined within the function's prototype and returned with the RETURN operation code.

```
*****
* Procedure Skeleton
*****
P MyFirstProc  B
*-----
FMyFile  IF  E      K DISK
*-----
D MyFirstProc  PI

D MyFileDS    E DS      ExtName(MyFile) Qualified
* Additional local Variables
*-----
/Free

//Source Code goes here

Return;
//-----
// Local Subroutines
//-----
/End-Free
P MyFirstProc  E
*****
* Function Skeleton
```

```

*****
P MyNextFnc    B
*-----
D MyNextFnc    PI      256A  Varying

D RtnVal      S      256A  Varying
* Additional local Variables
*-----
/Free

//Source Code goes here

    Return RtnVal;
/End-Free
P MyNextFnc    E

```

Example 1: Skeleton for a RPG procedure and function

The following example shows the source code for two exported functions to convert any Unicode text to either upper or lower case.

```

D constUpper   C          Const(%UCS2('ABCDEFGHIJKLMNOPQRSTU+
D              VWXYZ'))
D constLower   C          Const(%UCS2('abcdefghijklmnopqrstuvwxyz+
D              vwxyz'))
*****
* Purpose:      convert a Unicode string to lower case
*****
P Cvt2Lower    B          Export

D Cvt2Lower    PI      512C  Varying
D ParText      512C  Varying Options(*Trim) Const
*-----
/Free
    Return %XLate(ConstUpper: ConstLower: ParText);
/End-Free
P Cvt2Lower    E
*****
* Purpose:      convert a Unicode string to upper case
*****
P Cvt2Upper    B          Export

D Cvt2Upper    PI      512C  Varying
D ParText      512C  Varying Options(*Trim) Const
*-----
/Free
    Return %XLate(ConstLower: ConstUpper: ParText);
/End-Free
P Cvt2Upper    E

```

Example 2: Functions Cvt2Lower and Cvt2Upper to convert strings into upper/lower case

After having finished the source code, it must be compiled.

Module, the basic Building Block of all Programs and Service Programs

In an OPM world there was only a single source member written in a single programming language that was compiled into a program.

In an ILE world, a program can be composed of multiple building blocks that even may have been written in different

programming languages. Before those blocks can be put together, the source code must be compiled by using the appropriate language compiler. The result of this compilation is a non-executable object with the object type *MODULE. In other words a module is a piece of compiled source code that cannot be executed directly but must be bound to either a program or service program in an additional step.

And here is the real difference between the OPM and ILE methodology. An OPM program can be developed top down, but in an ILE world the building blocks must be developed first before they can be composed within a program. That means functions and procedures to be executed must be coded first, modules must be generated from the source code and finally the modules must be bound to programs. ILE development goes from the bottom (procedures / functions) to the top (program).

A module can be compiled with the CRTxxxMOD (Create Module) CL command, where xxx represents the programming language. There is a separate CL command for creating modules for any ILE language, i.e. RPG (-> CRTRPGMOD), CL (-> CRTCLMOD), for Cobol (-> CRTCLMOD) etc.

The following example shows the CL command for generating the module MYRPGMOD in the MYOBLIB library.

```
CRTRPGMOD MODULE(MYOBLIB/MYRPGMOD)
  SRCFILE(MYSRCLIB/QRPGLESRC)
```

Example 3: Generating a RPG module using the CRTRPGMOD command

Programs and Bind By Copy

An ILE program is composed of one or multiple modules, independent of which programming language the source code was originally written. At least one of the modules must have a main-procedure included, since only main-procedures can be called from a command line or a menu or can be submitted. A program cannot be composed of modules that solely contain internal and exported procedures.

A program is bound with the CRTPGM (Create Program) CL command. Within the CRTPGM command all modules to be bound to the program must be listed. The module with the main-procedure to be called first must be specified as Program Entry Procedure (PEP) module.

In the following example, the MYPROGRAM program is built by binding the MYPROGRAM, MYCLMOD, MYRPGMOD1 and MYRPGMOD2 modules together that were generated before. The MYPROGRAM module is specified as being the program entry point procedure module.

```
CRTPGM PGM(MYOBLIB/MYPROGRAM)
  MODULE(MYOBLIB/MYPROGRAM
    MYOBLIB/MYCLMOD
    MYOBLIB/MYRPGMOD1
    MYOBLIB/MYRPGMOD2)
  ENTMOD(MYOBLIB/MYPROGRAM)
```

Example 4: Generating a program by binding multiple modules together with the CRTPGM command

In the binder step all modules are physically copied into the program object, also referred as Bind By Copy. At runtime these in the program object included copies of the modules are executed. After a module is bound to the appropriate program(s) or service-program(s) it is no longer needed and can be deleted.

If a program consists only of a single module, it can be generated directly from a single source member by using the CRTBNDxxx (Create Bound Program) CL commands where xxx represents the programming language. Nevertheless, under the covers a two-step compilation is performed. This means that the module is created first in the QTEMP library, bound to a program with the same name, and then deleted afterwards.

When binding modules directly to program objects, a bind by copy is performed. If the same procedures must be called from multiple programs, the modules containing the procedures must be bound to each of these programs. If a procedure must be modified, the appropriate module must be replaced in all programs it is bound to. A module bound to a program can be replaced with the UPDPGM (Update Program) CL Command.

If a new module must be added to an existing program the UPDPGM command cannot be used, instead the program must be

recreated by rebinding all modules with the CRTPGM command.

Service-Programs and Bind By Reference

One of the main goals of the ILE concepts is to avoid duplicated source code. Reusable source code will be wrapped with exported procedures that can be called from elsewhere. A highly modularized application development may result in several hundred or even thousands of exported procedures located in as almost as many source members. Modifying a procedure or adding a new one to a module and rebinding/recreating all programs where the appropriate module is bound to might end up in being a real nightmare.

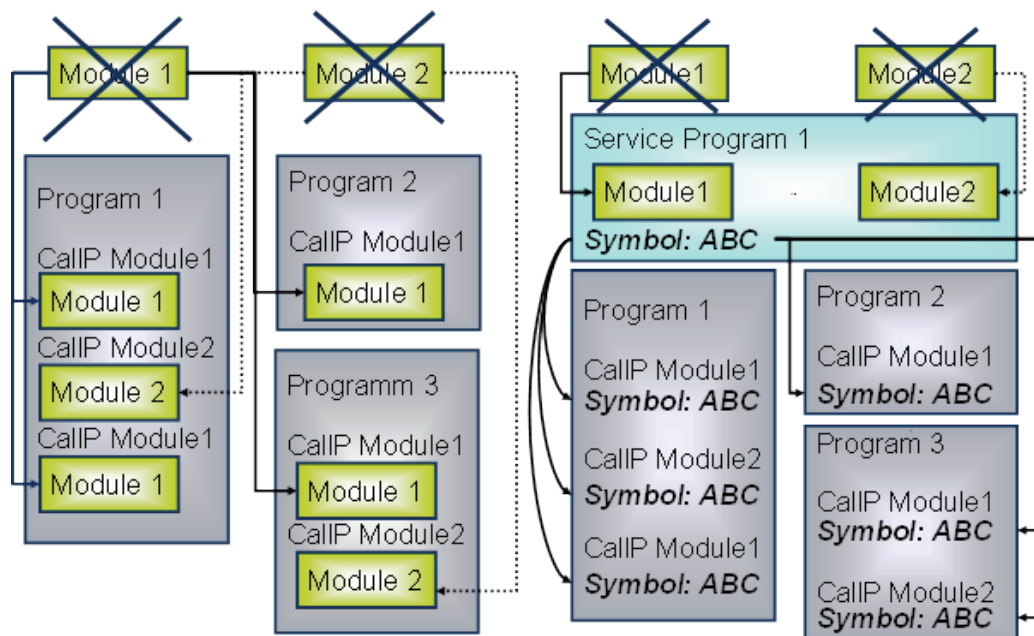
... and here is where service programs come into play.

Like a program, a service program is a compound of modules that are copied into the service-program object. A service program is bound with the CRTSRVPGM (Create Service Program). Before binding a service program all modules to be bound must be created. When executing the CRTSRVPGM command all modules have to be listed. After the modules are bound to the appropriate (service) program objects they are no longer needed and can be deleted.

Contrary to a program generation a module containing a main procedure is not required. Instead of having only a single program entry point a service-program has multiple entry points, one for each exported procedure, since an exported procedure can be called externally.

A service program is nothing other than a container to hold exported procedures. To be found at activation/run time each service program gets a unique signature or symbol.

When creating a program or service program where procedures located in different service programs are called, these service programs must be listed within the CRTPGM or CRTSRVPGM command, just as modules have to be. But instead of copying the appropriate service program (or its modules) physically into the caller (service) program object only the unique signature of the service program to be bound is included. This process is referred as Bind by Reference.



If a procedure within a service program must be modified, the module within the service program must be replaced. Since in the caller (service) programs only the unique signature of the service program (and not the module itself) is included, there is no need to rebind these (service) programs.

The following example shows the difference between bind by copy and bind by reference. On the left side two modules are bound by copy to three programs. If one of the modules has to be modified, the programs that include the module must be rebound. On the right side both modules are bound to a service program and the symbol/unique signature of the service program is bound to the programs. If one of the modules has to be modified it must be only replaced in a single place, i.e. within the service program. The programs that call procedures out of the service program stay untouched.

Example 5: Comparing Bind by Copy and Bind By Reference

Binding Directories for facilitating the Binder Step

As previously mentioned creating programs or service programs require all modules and/or service programs containing any procedure that is called to be listed. In a highly modularized application development this approach can be cumbersome.

To avoid listing all modules and service-programs to be bound to the program or service program in the CRTPGM or CRTSRVPGM command, binding directories can be used.

A binding directory is an object with the object type *BNDDIR created with the CRTBNDDIR (Create Binding Directory) -CL command. A binding directory contains a list of modules and/or service programs that can be bound to a program or service-program.

Modules and/or service programs can be added to or removed from the binding directory by executing either the ADDBNDDIRE (Add binding directory entry) or RMVBNDDIRE (Remove binding directory entry) CL commands.

If binding directories are specified in the binder step (with the CRTPGM or CRTSRVPGM command), these binding directories are searched to find the modules and/or service programs containing the called procedures. If an appropriate module is found it is bound by copy to the (service) program object. If a service program is found its unique signature is included in the (service) program object that means it is bound by reference. If the procedure is neither found in a module nor a service program listed in the binder directory, the binder step will fail.

The easiest way to work with binding directories is using a single binding directory containing all service programs for a specific application or library that is always referred within the binder step.

If multiple modules are bound to programs or service programs and the modules are deleted immediately after binding, the global binding directory could also contain modules. If the modules are not deleted, I'd suggest using different binding directories containing modules only to avoid problems with binding by copy instead of reference and/or problems with duplicate procedures.

If a program is created from a single source member, but is calling procedures located within multiple service programs, it can be compiled with the CRTBNDRPG command in composition with a binding directory. Prerequisite for using CRTBNDRPG is the program must not run within the default activation group. If the DFTACTGRP compile option is set to *NO, a single or multiple binding directories can be specified within the CRTBNDRPG command.

Alternatively, it is possible to include the activation group and binding directories in the H-Specs of the source code for the program. The H-Specs will override the options within the compile command, with the exception of the BNDDIR keyword. All binding directories specified within the H-Specs AND all binding directories specified within the compile command are searched.

The following code snippet shows the H-specs of a program. The binding directories MYBNDDIR and MYBNDDIR2 are searched to find the modules and/or service programs containing the called procedures. Because of the compiler directive the activation group information, the keywords DFTACTGRP and ACTGRP, is only considered if the program is compiled with the CRTBNDRPG command. If a two-step compilation is performed, i.e. CRTRPGMOD and CRTPGM the activation group information is ignored.

```
/If Defined (*CRTBNDRPG)
H DftActGrp(*No) ActGrp('MYPGM')
/EndIf
H BndDir('MYBNDDIR': 'MYBNDDIR2')
```

Example 6: H-Specs for generating a program searching binding directories

Binder Language to avoid Service Program Signature Violation

When using service programs the unique signature of the service program is included in all programs and service programs that call any procedure out of this service program. The unique signature is (automatically) determined based on the list of procedure and data item names to be exported and from the order in which they are specified. That means if a procedure must be added to an existing service program the unique signature of this service program will change and all of programs and service programs that call any procedure from this service program must be rebound.

... not much better than binding modules directly?!

... but wait there is the binder language!

The binder language allows the signature to be explicitly predefined by the programmer. The signature can also be generated automatically, but in composition with the binder language the previous signatures can be kept within the service program objects. In both cases rebinding all dependent programs and service programs is not required after having added a procedure to an existing service program.

The source code containing binder language commands is stored in source physical file members with the attribute BND. The binder language consists only of 3 commands that build an export block

- **STRPGMEXP – Start Export Block**

The STRPGMEXP command identifies the start of the export block.

Within the STRPGMEXP the signature of the Service Program can be either explicitly predefined or generated automatically. If an explicit signature is specified, only a single export block, containing the currently exported procedures is required. New exports can be added to the end of the list of exports.

If the signature is automatically generated (*GEN), the old and all other previous export blocks must be kept within the source member. Based on this export blocks, the current and all previous signatures are managed and included in the service program object. At activation time the service program is located based on any of these signatures depending on what signature is stored in the caller (service-) program object.

- **EXPORT – Export Procedure**

The export command must be specified for each procedure that has to be exported from the service program. Only procedures that are listed within the export block can be called from outside the service program. In other words, a procedure with the keyword EXPORT within the beginning P-Statement that is not listed within the EXOPRT list cannot be called from outside.

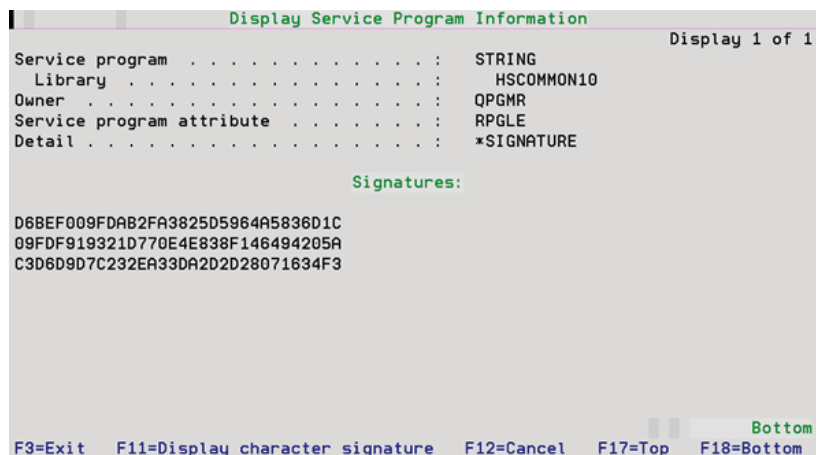
- **ENDPGMEXP – End Export Block**

The ENDPGMEXP command ends the export block.

When binding a service program, the source member containing the binder language can be specified within the CRTSRVPGM command.

The following example shows the binder language source code for the service program STRING with an automatically generated signature. First the service program contained only 3 exported procedures, CVT2LOWER, CVT2UPPER and CVT2MIXED. Later the procedures RTVSTRINGPOSINFOCASE and RTVSTRINGPOSINFO were added. With the last enhancements the procedures TEXTSPACING and TEXTSPACINGUPPER were integrated.

```
STRPGMEXP PGMLVL(*CURRENT)
EXPORT  SYMBOL(CVT2LOWER)
EXPORT  SYMBOL(CVT2UPPER)
EXPORT  SYMBOL(CVT2MIXED)
EXPORT  SYMBOL(RTVSTRINGPOSINFOCASE)
EXPORT  SYMBOL(RTVSTRINGPOSINFO)
EXPORT  SYMBOL(TEXTSPACING)
EXPORT  SYMBOL(TEXTSPACINGUPPER)
ENDPGMEXP
```



```
Display Service Program Information                                     Display 1 of 1
Service program . . . . . : STRING
Library . . . . . : HSCOMMON10
Owner . . . . . : QPGMR
Service program attribute . . . . . : RPGLE
Detail . . . . . : *SIGNATURE

Signatures:

D6BEF009FDAB2FA3825D5964A5836D1C
09FDF919321D770E4E838F146494205A
C3D6D9D7C232EA33DA2D2D28071634F3

F3=Exit  F11=Display character signature  F12=Cancel  F17=Top  F18=Bottom
```



```

STRPGMEXP PGMLVL(*PRV)
EXPORT  SYMBOL(CVT2LOWER)
EXPORT  SYMBOL(CVT2UPPER)
EXPORT  SYMBOL(CVT2MIXED)
EXPORT  SYMBOL(RTVSTRINGPOSINFOCASE)
EXPORT  SYMBOL(RTVSTRINGPOSINFO)
ENDPGMEXP

```

```

STRPGMEXP PGMLVL(*PRV)
EXPORT  SYMBOL(CVT2LOWER)
EXPORT  SYMBOL(CVT2UPPER)
EXPORT  SYMBOL(CVT2MIXED)
ENDPGMEXP

```

Example 7: Binder Language with automatically generated signature

The following screen shot shows the automatically generated and the service program STRING integrated signatures.

Example 8: Service Program with multiple automatically generated signatures

Activation Groups

All ILE programs and service programs are activated within a substructure of a job called an activation group. Activation Groups are temporary storage structures within the job that include all resources necessary to run the programs, such as static and automatic variables, dynamically allocated storage, open data paths and some error routines.

All programs and service programs run in an activation group. The activation group is associated with the program or service program within the binder step, i.e. it can be specified within the CRTPGM, CRTSRVPGM and CRTBNDRPG commands.

- **Default Activation Group**

All OPM programs and ILE programs compiled for the default activation group run in the default activation group. ILE programs and/or service programs generated with the activation group *CALLER that are called from a program or service program running within the default activation group are also activated in the default activation group.

The default activation group is automatically started with the job and will be automatically ended when the job ends. A default activation group cannot be terminated manually.

- ***NEW**

Each time a program generated with activation group *NEW a new activation group is built. The activation group is automatically ended as soon as the program ends, i.e. all allocated resources are freed.

Activation group *NEW may be preferable for programs that are only executed once a day in an interactive job. Before linear main procedures were introduced only programs running in a *NEW activation group could be called recursively.

- ***CALLER**

The program or service program is activated within the same activation group as the caller.

Activation group *CALLER is the default when binding service programs and should be used in either way for all service programs containing INSERT, UPDATE and/or DELETE routines, especially when working with commitment control. The default value for commitment scope within the STRCMTCTL (Start commitment control) command is *ACTGRP. Using the default commitment scope a transaction spread over multiple activation groups may cause problems especially when a ROLLBACK action must be performed.

- **Named Activation Group**

Any name can be specified as activation group name when binding the program or service program. When calling the program or service program an activation group with the specified name is generated. A named activation group remains existent until the job ends or the activation group is explicitly ended with the RCLACTGRP (Reclaim Activation Group) CL Command.

A named activation group should be used for all ILE programs that are executed more than once and could be used for all service programs containing only universally applicable procedures (for example date or string procedures).

The activation group in which a program or service program must be activated can be specified in all creation and update commands, such as CRTBNDxxx, CRTPGM, UPDPM, CRTSRVPM and UPDSRVPM.

Additional Links, Tutorials and Articles

In order to prepare for the exam you will find a lot of valuable information under the following links:

- **ILE RPG Language Reference**
<http://publib.boulder.ibm.com/infocenter/series/v7r1m0/index.jsp?topic=%2Frzahg%2Frzahgrpplangref.htm>
- **Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More**
<http://www.redbooks.ibm.com/abstracts/sg245402.html?Open>
This redbook provides among other useful information a really good and easy to understand introduction into the ILE concepts and embedded SQL.
- **ILE Concepts**
<http://publib.boulder.ibm.com/infocenter/series/v7r1m0/index.jsp?topic=%2Frzahg%2Frzahgileconcept.htm>
- **DeveloperWorks – RPG Café**
<https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=b542d3ac-0785-4b6f-8e53-f72051460822>
- **IBMSystems Magazine – IBM i – Developer - RPG**
<http://www.ibmsystemsmag.com/ibmi/developer/rpg/>
You'll find a lot of articles written by Jon Paris, Susan Gantner and others
- **IT Jungle – The Four Hundred Guru**
<http://www.itjungle.com/fhg/fhgindex.html>
- **MC-Press Online – Programming - RPG**
<http://www.mcpressonline.com/programming/rpg/>
- **iPro Developer (formerly System iNetwork)**
<http://www.iprodeveloper.com/archives/magazines/ipro-developer>
iPro Developer provides a magazine and several newsletters. You need a subscription and not all articles are for free.
- **Scott Klement's Webpage**
<http://www.scottklement.com/>

IBM i Access for Training

If you need access to an IBM i for training, writing your own example programs, testing purposes and preparing for the exam, you may contact: RZKH – Rechenzentrum Kreuznach (Holger Scherer)

RZKH offers free (Release V5R3) or paid (higher releases) access to an IBM i system.

<http://www.rzkh.de/cgi-bin/db2www/getaccounteng.d2w/main>

And now have fun in preparing for the ILE RPG certification exam!

About the Author:



Birgitta Hauser has been a Software Engineer since 2008, focusing on RPG, SQL and Web development on System i/Power i at Toolmaker GmbH in Germany. She graduated with a business economics diploma, and started programming on the AS/400 in 1992. Before joining Toolmaker, she was responsible for the complete RPG, ILE, and Database programming conceptions for a software house with its own Warehouse Management Software Package. She also works in consulting and education as a trainer for RPG and SQL developers. Since 2002 she has frequently spoken at the COMMON User Groups in Germany, Switzerland and USA. In addition, she is co-author of two redbooks and also the author of several papers focusing on RPG and SQL for IBM Developer Works and a German magazine.

Thanks for reading & good luck!