/Free Your RPG

Susan Gantner
susan.gantner@partner400.com
www.Partner400.com
www.SystemiDeveloper.com

If you thought that RPGIV had changed over the years - well "You ain't seen nothin' yet!!"

Most of us have been making extensive use of Evals since they were first introduced with RPG IV in V3R1. Perhaps like us you have experienced frustration when forced to split a line because you had run out of space! Maybe you were even tempted to shorten that nice meaningful variable name to avoid having to use a second line! While you were contemplating this dilemma, you might have also noted the fact that there was a huge area of white space to the left of the Eval you couldn't use.

V5R1 put an end to that frustration by introducing the notion of completely free-format calculation specs. Coupled with a large number of new Built-In Functions (BIFs) this "New" RPG remains familiar, while offering some very powerful new capabilities.

In this session we will look at:

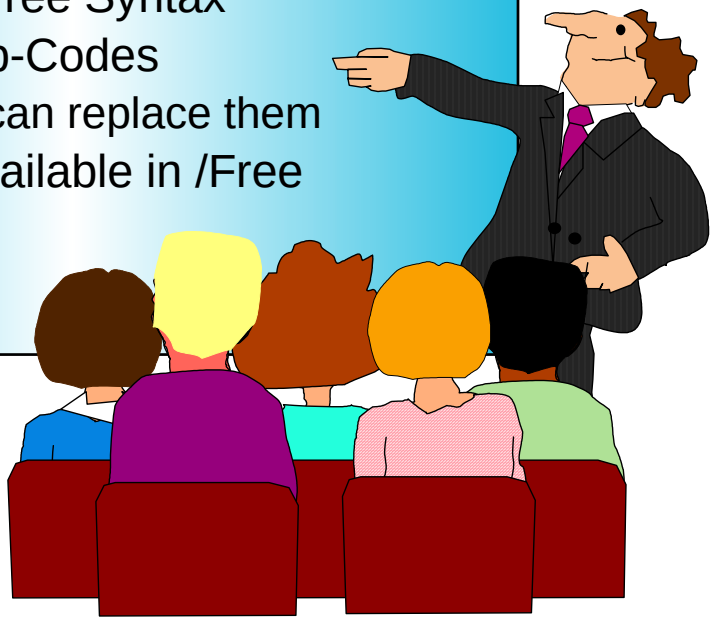
- How to code freeform RPG
- The new BIFs that add power to the language
- New functions that only work in /Free form logic

If you have any questions, please feel free to contact the author: Susan.Gantner@Partner400.com

Agenda

Partner400

- Why /Free?
- The Basics of /Free Syntax
- Unsupported Op-Codes
  - And BIFs that can replace them
- Options Only available in /Free



Why /Free?

Partner400

1. Indented logic is more understandable

- Match up nested If ... EndIf, DoW ... EndDo, etc.

2. More efficient use of space in source code

- No more wasted space between C and Op Code column
- Complex expressions take fewer lines

3. More room for larger names

- More descriptive and/or qualified data names

4. New features in /Free only

- Freed from the limitations of 2 Factors and a Result field

5. Attraction of new RPG programmers

- /Free looks and feels more like other modern languages
- Often more powerful for business applications

1) When writing free format logic, I can indent the statements to show the structure of the code. The code is more easily understood, especially in cases involving more complicated logic blocks, such as nested If statements, Select/When statements or Monitor blocks. The easier the code is to understand, the faster and less error prone it will be to maintain.

2) I find that I seldom use Factor 1 in RPG IV. At the same time, because I am no longer limited to simple arithmetic operations, many expressions - especially those involving built-in functions - get quite lengthy, often requiring multiple lines to complete. Therefore with fixed format logic, the source edit screen typically has a huge amount of unused space on the left (i.e., where Factor 1 and the outdated left-hand conditioning and cycle control indicators would go) while the right side is complicated with many multi-line statements. Using free format logic, I have much more space for my expressions. My edit screens are "cleaner" and I can see more logic at a time because there are fewer multi-line statements.

3) I have fewer length limitations for names in free format logic. I don't often set out to create COBOL-like names of 30 characters. On the other hand, it's nice that I'm not required to think of an acceptable abbreviation for "ProcessCustomer" or "ErrorMsgDisplay", which are just slightly too long to fit into a fixed format column. Qualified data names allow powerful features such as data structure arrays and nested data structures, but cannot practically fit into a 14 character Factor 1 column.

4) Some new language functions are only available in free format logic. This is because a column-oriented language such as traditional RPG puts severe limitations on some kinds of new features that require significant space in the statement for implementation.

5) Last, but not least, free format logic brings RPG more in line with other modern programming languages, all of which use free format logic. This is important for attracting new developers coming into the marketplace. Traditional fixed format RPG less attractive and gives RPG the undeserved appearance of an old-fashioned language not up to modern application tasks. RPG is a powerful and flexible language that many young developers come to prefer over other more popular language options for business applications. But they must first be attracted to learn the language.

```
C                   READ(E)   TransFile
C                   DOW       Not %EOF(TransFile)
C                   IF        %Error
C                   Eval      Msg = 'Read Failure'
C                   LEAVE
C                   ELSE
C     CustNo        CHAIN(N)  CustMast      CustData
C                   Eval      CustName = %Xlate(UpperCase:LowerCase
C                                        : CustName)
C                   EXSR      CalcDividend
C                   READ(E)   TransFile
C                   ENDIF
C                   ENDDO
```

```
READ(E) TransFile;
DOW not %EOF(TransFile);
   IF %Error;
      Eval Msg = 'Read Failure';
      LEAVE;
   ELSE;
      CHAIN(N) CustNo CustMast CustData;
      Eval CustName = %Xlate(UpperCase : LowerCase : CustName);
      EXSR CalcDividend;
      READ(E) TransFile;
   ENDIF;
ENDDO;
```

This small example of a comparison of fixed format RPG IV and /Free format logic illustrates some of the advantages described on the earlier chart.

Free Form C-Specs

Calculation Specs can now be completely freeform

- Freeform specs are introduced by a /FREE entry
- /END-FREE is used to return to fixed format
- You can mix /Free and Fixed .... but don't do it

Syntax: **Op-Code{(Exten)} { Factor1 } { Factor2 } { Result Field };**

- Operands no longer limited to 14 characters
- Each statement must end with a semicolon (;)
- Only 1 Op-code can be coded on a line
- Not all Fixed form Op-codes are supported


```
/Free
   If CustomerNumber = *Blanks;
      Eval CustomerError = True;
    Else;
      CustomerError = False;
   EndIf;
/End-Free
```

Although freeform can be mixed with traditional C specs, the result is code that is ugly and hard to maintain. Avoid it like the plague!

Wherever possible you should replace operations that do not have a direct freeform equivalent with options that will work in free form instead. The degree to which such replacement is "possible" depends on which release of the operating system you are writing for. We will be discussing the replacement options later in the session.

One of the really nice aspects of using /Free form calcs is that you can indent your source statements to highlight the logic flow as shown in this short example.

Note the use of the semicolon (;) to specify the end of the statement. As you will soon discover, if you forget it, you will get some really interesting error messages!

As shown in the example, some op-codes can be omitted completely. In the example we have omitted the EVAL op-code In the assignment CustomerError = False. Since the EVAL is optional, either option will work. Later we will look at the exception to this rule.

By the way - the only "Convert to /Free form" option supplied by IBM is provided within the CODE and WDSc editors. And that conversion is a very rudimentary one. Other software vendors, such as Linoma Software offer much more full-function conversion options.

/Free Form Calcs Rules

The Opcode & operands can appear anywhere in posns 8 - 80

- Positions 6 & 7 must be blank

Comments are delimited by //

- And may be placed at the end of any free form statement (after the ;)
- The old-style (*) comments are not accepted

There is no way to define fields in /Free form specifications

- But then you wouldn't want to do such a naughty thing anyway!

Resulting indicators may not be specified

- The BIFs such as %FOUND should be used instead
  - More on this later

Two Op Codes are optional: Eval and CallP

If you are not in a position to move to V5's /Free format yet, there are a number of steps you can take to prepare yourself for the change. These are all good programming style anyway so you can't lose!

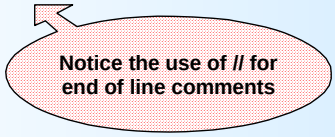
Use only those opcodes supported by freeform
Defining all variables on D-Specs
Do NOT use conditioning indicators
Avoid resulting indicators whenever possible.

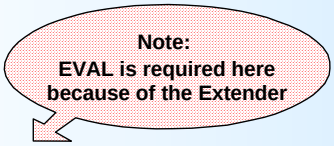
Free-form Example

- Note that subroutines can be defined and invoked in /Free form

```
/FREE
  READ(E) TransFile;         // Read file to prime do loop
  DOW not %EOF(TransFile);   // Continue until all records processed
      IF %Error;
          DSPLY 'The read failed';
          LEAVE;
      ELSE;
          CHAIN(N) CustNo CustMast CustData;
          CustName = %Xlate(Upper : Lower : CustName);
          EXSR CalcDividend;
          READ TransFile;
      ENDIF;
  ENDDO;

  BEGSR CalcDividend;
    TotalSales = %XFoot(MthSales);
    EVAL(H) Dividend = TotalSales / 100 * DivPerc;
    Record_transaction();
  ENDSR;
```





Note again the benefit of indentation of the source. With the source coded this way, it is very clear which statements belong to the If block and the Else block. And it is clear that that the IF/ELSE block is all part of the DOW.

Notice that we must use the %EOF and %Error built-in functions because resulting indicators cannot be used in freeform calcs. That's OK, because even in fixed format source, the use of the BIFs makes the code far more readable and understandable.

Notice that in this code sample, the EVAL operation code has been omitted - except where it was necessary to include it because of the half adjust extender.

Now for a tough question: Assuming these calcs are syntactically correct (i.e. the program containing these calcs will compile), what could Record_transaction() be? Is it an array element? A subprocedure call? A program call?

It is either an subprocedure or a program call (or, to be more specific, it is the name on the prototype given to a subprocedure or program to be called.) The CALLP (call with prototype) operation code, like the EVAL, can be omitted, as it is in this case.

You see another change in syntax for Version 5 in that same line, because prior to V5, if the procedure or program required no parameters to be passed, we could NOT have coded the empty parentheses. It is a good idea to use the empty parentheses notation so the name is not mistaken for a field name. In a free form spec, it is required to use the empty parentheses when no parameters are being passed.

Op-codes not supported in /Free

These fall into six main categories

- "Old fashioned" Op-codes whose use is discouraged
  - e.g. ANDXX, COMPxx, DEFINE, DOxyy, GOTO, and IFxx
- Those for which new support has been provided
  - e.g. ALLOC, CHECK, EXTRCT, LOOKUP, TIME and XLATE
- Op-codes with existing (if not identical) expression support
  - e.g. ADD, DIV, MULT and SUB
- Those supported by a combination of enhanced and new support
  - e.g. ADDDUR and SUBDUR
- "Problem Children"
  - KFLD and KLIST (Alternative support for these was added in V5R2)
- The ones the compiler writers don't like: MOVE, MOVEL, MOVEA
  - EVAL is the alternative but does not directly support type transforms
    - E.g. Alpha to Numeric, Numeric to Alpha
  - Function has been added to existing BIFs to support this
    - But you need to be on V5R3 before you have a complete solution

The next 2 charts contain some substitutions for op codes that are not supported in /Free format.

Note that the Op-code substitutions are not necessarily one-to-one. The substitute might not work exactly as the original code, especially in the area of error handling. For example the default for numeric overflow on an ADD operation is to truncate the result and ignore the error. The default for an addition in an EVAL type operation is to blow up! You need to bear this in mind if you are converting. Personally we would rather know if numeric overflow occurs, which is why we have always preferred using EVAL to the older style operations.

Op-Code Replacements

Op-Code	/FREE Substitute
ADD	Operator +
ADDDUR	Operator +
ALLOC	BIF %ALLOC
ANDxx	Logical operator AND
BITxx	(Bitwise operations in V5R2)
CABxx	(see GOTO)
CALL	Op-code CALLP
CALLB	Op-code CALLP
CASxx	Op-code IF with EXSR
CAT	Operator +
CHECK	BIF %CHECK
CHECKR	BIF %CHECKR
COMP	Operators =, <, >, etc.
DEFINE	LIKE or DTAARA on D-Spec
DIV	Operator / or BIF %DIV

Op-Code	/FREE Substitute
DOUxx	Opcode DOU
DOWxx	Opcode DOW
END	Opcodes ENDDO, ENDIF, etc.
EXTRCT	BIF %EXTRACT
GOTO	LEAVE, ITER, LEAVESR, etc.
IFxx	IF
KFLD	(See KLIST)
KLIST	(Various options in V5R2)
LOOKUP	%LOOKUPxx or %TLOOKUPxx
MxxZO	(Bitwise operations in V5R2)
MOVE	EVALR or %DATE, %TIME, etc.
MOVEA	%SubArr in V5R3 or …..
MOVEL	EVAL or BIFs for Date, Time etc.
MULT	Operator *
MVR	BIF %REM

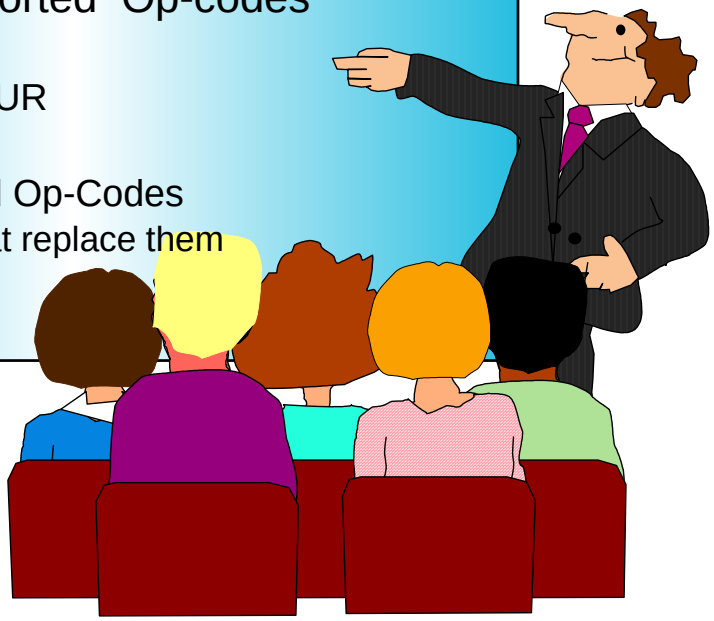
Op-Code Replacements

Op-Code	/FREE Substitute
OCCUR	BIF %OCCUR
ORxx	Operator OR
PARM	(see PLIST)
PLIST	D-Spec PI & PR definitions
REALLOC	BIF %REALLOC
SCAN	BIF %SCAN
SETON	EVAL *lnxx = *ON
SETOFF	EVAL *lnxx = *OFF
SHTDN	BIF %SHTDN
SQRT	BIF %SQRT
SUB	Operator -
SUBDUR	Operator - or BIF %DIFF

Op-Code	/FREE Substitute
SUBST	BIF %SUBST
TAG	(see GOTO)
TESTB	(Bitwise ops in V5R2)
TESTN	(Bitwise ops in V5R2)
TESTZ	(Bitwise ops in V5R2)
TIME	Time & Timestamp BIFs
WHENxx	Opcode WHEN
XFOOT	BIF %XFOOT
XLATE	BIF %XLATE
Z-ADD	Opcode EVAL
Z-SUB	Opcode EVAL

Opcode Alternatives

Performing I/O operations in /Free
Replacing unsupported Op-codes
- MOVE
- ADDDUR & SUBDUR
- CALL
- Other Unsupported Op-Codes
  - And the things that replace them



The new BIFs are not limited to /Free but can also be used in Fixed-form Evals

For most people the biggest "omission" from the list of op-codes supported in /Free is MOVE. There have been Internet flame wars raging on and off for over two years on this topic - but IBM show no sign of changing their minds.

We don't have time in this session to go into details about all the of the /Free substitute methods, but we will take a look at some of the less obvious and/or most required ones.

Converting I/O Operations to /Free

You cannot use numbered indicators so ...

- Replace them with the I/O related BIFs first introduced in V4R2
  - They return *On or *Off based on the last relevant I/O operation to the file

%EOF (FileName)

- Set by: READ, READE, READP, READPE & WRITE to subfile

%EQUAL (FileName)

- Set by SETLL if an exact match is found

%FOUND (FileName)

- Note that this is the Inverse of using the NR indicator
- Set by: CHAIN, DELETE, SETGT, SETLL

```
Read TransFile;
  If Not %EOF(TransFile);
    Chain CustNum CustFile;
    If %Found(CustFile);
      SetLL CustNum Invoices;
        If %Equal(Invoices);
          ............
```

You haven't had to use resulting indicators on I/O operations since V4R2, which is when these BIFs were first introduced. /Free gives you no choice. There are no resulting indicators, so you have to use these BIFs.

The use of the file name on the %FOUND and %EQUAL built-in functions is optional, but highly recommended. If you don't specify a file name, the "naked" BIF defaults to supplying the result from the last file that could have set this condition. This is probably OK when you first write the program - but later during maintenance it would be very easy for other lines of code to be introduced, perhaps something as simple as a call to a subprocedure or subroutine. If that code performs an I/O operation, then the result you will be testing may have nothing to do with the file you think you are testing!

Note that %FOUND is not an exact match to its corresponding resulting indicator. Rather it is the reversal of the indicator setting. When the indicator would have been on, %FOUND will be off. When the indicator would have been off, %FOUND will be on. The reason for this is obvious if you think about it. If the compiler writers had matched the indicator exactly, the function would have to be called %NOTFOUND. Suppose that you wanted to write a test to see if a record was found by a CHAIN operation. You would have had to code IF NOT %NOTFOUND and that would be too silly for words!

There is another BIF in this set which provides functionality that was not available before. This is %OPEN. This provides an easy way for a programmer to determine if a file has been opened. %OPEN is the only one of the new built-ins that requires the use of a file name.

The file built-in functions always reflect the results of the last file operations to a file even if the corresponding resulting indicator is specified. For example, %FOUND(file) is updated even if the last CHAIN operation had a "no record" resulting indicator coded.

Any one I/O operation will only set the values for the built-in functions that are relevant to the operation. For example, the DELETE operation will not affect the value of the %EOF built-in function for that file.

Associated Error Handling Options

The (E)rror Op-code extender

- Can be used with any op-code that allow an error indicator
  - Affects error handling in exactly the same way as coding the error indicator

%ERROR

- Returns the same value as the Error indicator
- Value is always set to *Off before the operation begins
- Normally used with %Status to determine the exact cause

%STATUS(FileName)

- Returns 0 (zero) if no error has occurred since the beginning of the last operation that specified the (E) extender
  - Otherwise, the most recent program or file status value
- Returns the same value as the *STATUS field in the INFDS or PSDS
  - For I/O errors and program errors respectively

These options apply to all op-codes with an error indicator

Another option - the Monitor operation . . .

Note unlike most of the other I/O built-ins, %Error cannot specify a file name. %Error always reflects the state of the last I/O operation for which the (E) extender was specified.

The advent of the new built-in %STATUS means that many programs that only used to define the INFDS for the purposes of defining the file status no longer need to do so. The resulting code is often far easier to read.

Note that the (E) extender can be used with any op-code that can define an error indicator, it is not restricted to I/O operations. Of these, the most commonly used is the TEST op-code. We will look at the usage of this when we discuss dates later.

Replace MOVE - Character to Numeric

Use %DEC (expression { : precision : decimals })

- Expression can now be character or numeric
- This capability also applies to %FLOAT, %INT, and %UNS
- As well as the Half adjust variants (%DECH, %INTH, and %UNSH)

Exception (Status 105) is signalled if character field not valid

- Use MONITOR to catch this error
  - See Rules and comments on "clean up" on Notes page

```
D CharField       S             12a   Inz(' -12345.678')
D NumField        S              9p 2
D Length          C                   %LEN(NumField)
D Decimals        C                   %DECPOS(NumField)

 /Free
   NumField = %DecH( CharField : 9 : 2 );
 // This version "self adjusts" if the definition of NumField changes
   NumField = %DecH( CharField: Length: Decimals);
```

Rules for format of character string:
Sign is optional. It can be '+' or '-', leading or trailing
The decimal point is optional
Blanks are allowed anywhere in the data.
For %DEC(H) both the Precision and Decimal Place parameters are compulsory
Floating point data (e.g. '2.5E4') is not allowed

In the example below, the assignment at <A> will work correctly. <B> though will cause an exception (Status code 105) because of the dollar ($) sign. IBM suggests a nice technique to handle this situation, as demonstrated at <C>. Simply put, the %XLATE BIF is applied to the character expression, substituting blanks for all unwanted characters (our example only replaces $ and , (comma). The resulting blanks will be ignored while processing the expression.
This deals with most common situations, but will not remove all sources of error. (e.g. there might be two decimal points in the field). One way to validate the character string is to use the subprocedure ChkNbr, which is part of the CGIDEV2 library available free from IBM. Go to www-922.ibm.com

If you do not wish to validate every character field before converting it, "Wrap" the operation in a MONITOR group (another wonderful V5R1 innovation) and catch the errors if and when they happen.

```
     D CharField1      S             14a   Inz('     1,525.95-')
     D CharField2      S             14a   Inz('    $1,525.95-')
     D NumField        S              9  2
     D NumberEdits     C                   '$,'
      /Free
<A>    NumField = %Dec(CharField1 : 9 : 2);
<B>    NumField = %Dec(CharField2 : 9 : 2);
<C>    NumField = %Dec( %Xlate(CharField2 : '$,' : '  ') : 9 : 2);
```

Replace MOVE/MOVEL - Char to Char

If fields are the same length

- Replacement is EVAL

If fields are of different lengths

- Replacement is (if blank padding is OK):
  - EVAL for MOVEL
  - EVALR for MOVE
- However, if padding is not desired
  - %Replace and/or %Subst will be needed
  - Or use a Data Structure

Handling for character to character moves depends on the length of both the from and to fields.

Very often you will see code that moves blanks to a field and then moves another value in. Sometimes this is done even when the two fields are both character fields and the same length! At one point in time they might have been different lengths - but nobody removed the blank fill when the field length changed.

In fact the "P" extender (i.e. MOVE(P) or MOVEL(P)) to cause target fields to be blank filled has been around for some time and most blank fills could have been removed years ago!

Replace MOVE - Numeric to Character

Basic option - no decimal places

- First choice is to use the BIF %CHAR
  - But only if you know that there will be no leading zeros
- Alternative is to use %EDITC with the 'X' edit code

If the numeric field contains decimal places

- Use %EDITC with the 'X' edit code
  - You need to study exactly what you are trying to achieve

The BIFs will produce a character string

- Once the BIF has been selected and you know the length of the result apply the same rules as for Character to Character

```
charField = %CHAR(numField);
charField = %EDITC(numField: 'X');
```

It is worthwhile when replacing MOVEs to determine exactly what was the intent of the original operation.

Note that %Char converts numeric fields to character, but it suppresses leading zeroes. If you want to retain leading zeroes in the character field, use %EDITC instead.

MOVE - Date to Numeric

%DEC (date | time | timestamp { : format })

- Length of returned value is the number of digits in the Date
  - Or Time or Timestamp
- e.g. %DEC(date : *MDY) = length of 6
- Length of %DEC(Timestamp) is always 20

If format is not specified - the format of the date field is used

- In the example below it would have been *ISO and used 8 digits

```
D mmddyy          S              6S 0

 * DatFld uses the default (*ISO) format
D DateFld         S               D   Inz(D'2003-06-27')

 * Convert DatFld to numeric using MOVE
C     *MDY          Move      DateFld       mmddyy

 // Equivalent code /Free code at V5R3
  mmddyy = %Dec(DateFld: *MDY);

 // Result:  mmddyy = 062703
```

V5R3!

This latest enhancement to %DEC takes care of one of the last "uglies" that were forced on us by the nonsupport of MOVE if /Free form.

In V5R2 IBM gave us the ability to use %DEC to convert character strings to numeric. This still left a problem with dates. To go from a character or numeric field to a date was not a problem - %Date gave us that in V5R1. But to go from date to number required the ugly:

Number = %DEC(%CHAR ..... combination as shown in the example below. Thankfully that has disappeared with the direct support for dates added in V5R3.

** Note that all of the above applies to %UNS, %INT and all of the half adjust variants as well. Not sure why you would want to half adjust a date but .... <grin>

Since many of you will not be using V5R3, here's a version of the sample code that works for V5R2.

```
// If you have to compile for V5R2 then this is the equivalent
mmddyy = %dec(%Char(DateFld: *mdy): 6: 0 );
```

If you need to do this in /Free form at V5R1 it gets pretty ugly, so we won't show you here. Personally we'd probably use %Char and a DS - but there are other methods including the use of C functions. Write us if you want to know more about them.

V5R3!

MOVE - Character/Numeric to Date

New BIF %DATE ({ expression { : date format } })

- Converts both character and numeric fields to date
- The second parameter specifies the format of the source field
  - Just as it did with MOVE
- If no parameters are passed then the system date is returned

Companion BIFs are %TIME and %TIMESTAMP

```
D CharDate1       S               6A   Inz('011549')
D CharDate2       S               6A   Inz('031954')
D ISODate         S                D   DatFmt(*ISO)
D USADate         S                D   DatFmt(*USA)

 * Loading a date field from a character field prior to V5R1
C     *MDY0         MOVE      CharDate1     ISODate
C     *MDY0         MOVE      CharDate2     USADate

  // The equivalent code in /Free
   ISODate = %Date(CharDate1: *MDY0);
   USADate = %Date(CharDate2: *MDY0);
  // Expressions are supported - e.g. Join 3 separate character fields
   USADate = %Date( Day + Month + Year): *DMY0);
```

The %DATE, %TIME and %TIMESTAMP BIFs are used to remove the requirement for a MOVE or MOVEL operation to convert data from character or numeric fields to date/time/timestamp fields.

Much like %CHAR will format date data into a character field, %DATE will do the reverse for character fields. The second (format) parameter specifies the format of the data being converted (i.e. the data represented by the first parameter). If the format is not specified, the default format for the program is used. If you recall, the default format for the program is the format specified with DATFMT (or TIMFMT for time fields) on the H spec or, if nothing is specified on the H spec, it will be *ISO.

Note that the format of the date (or time) returned will always be *ISO.

If you specify either *DATE or UDATE (to retrieve the job date) as the first parameter you should omit the format parameter altogether.

Note the difference between the job date and system date. *DATE or UDATE return the job date, %DATE() returns the system date (i.e. the current date). Also note that initializing a date field to *SYS using the D spec keyword only sets the INITIAL value for the field. So if a program runs over midnight, the initialized value in a date field defined with INZ(*SYS) will be different from that returned by %DATE().

Dates - Replacing SUBDUR

Calculating Durations

%DIFF(Date1 : Date2 : DurationType)

- Calculates durations between date, time or timestamp values
- Other duration calculations are performed by simple + and - operators
  - In conjunction with new duration Built-Ins

Duration types are as per ADDDUR / SUBDUR

- That is *MS, *S, *MN, *H, *D, *M, *Y
  - And the long-winded versions *MSECONDS, *SECONDS, *MINUTES, etc.

```
 * Is the loan due within the next 6 months ?
C     DueDate       SUBDUR    Today         MonthsToRun:*M
C                   IF        MonthsToRun < 6

 /Free
   MonthsToRun = %Diff(DueDate : Today : *M );
   If MonthsToRun < 6;

   If %Diff(DueDate : Today : *M ) < 6; // Alternate coding
```

Hurray! Courtesy of the /Free support, we finally have date duration support in expressions! This is a feature that RPG IV programmers have been wanting since V3R1. Most RPG IV programmers quickly became addicted to the extended factor 2 operation codes (e.g., EVAL, IF, etc.) and were dismayed to discover that in order to use the powerful date duration support built in to RPG IV from the beginning, they were forced to revert to the Factor 1, Factor 2, etc. format to use the powerful date duration operations.

The code examples here illustrate the ability to replace the ADDDUR and SUBDUR operation codes with expressions using either %DIFF or a simple arithmetic add (+) operation in combination with the duration BIF (%Years, in this example).

It also demonstrates that, because the new support is through BIFs, it can be used directly in expressions without the need for creating intermediate results.

On the next chart, we will show how to also replace the requirement for a MOVE operation code to get numeric data into a date or time field.

Now, all date data type operations can be done with free form operation codes.

Dates - Replacing ADDDUR / SUBDUR

Adding durations to a date, time or timestamp

- This function can now be performed by simple addition
- The durations are supplied via the appropriate BIFs
  - e.g. %MINUTES, %HOURS, %DAYS, %MONTHS, %YEARS, etc.
- Multiple durations can be handled in a single statement

Subtracting a duration works the same way

- i.e. It is achieved through the use of simple subtraction

```
 * These original duration calculations
C     ContrDate     AddDur    Cyears:*Y     ExpDate
C     ExpDate       AddDur    CMonths:*M    ExpDate
C     ExpDate       AddDur    1:*D          ExpDate

C     ExpDate       SubDur    90:*D         WarnDate

  // Can be replaced by these /Free calculations
  ExpDate = ContrDate + %Years(CYears) + %Months(CMonths) + %Days(1);

  WarnDate = ExpDate - %Days(90);
```

As on the previous chart, we are demonstrating here that since the new functionality is supplied by BIFs, they can be combined in a single expression - no need for the multiple and potentially error prone intermediate steps.

Combining The New Date Functions

The full power of the new BIFs appears when combining them

- In the example below it has significantly reduced the amount of code
  - The difference is actually greater than shown since in the new version there is no need to define the work fields Temp, Today and WorkDate
- %Date is used twice in the calculation
  - The first time to obtain the system date
  - The second to convert the numeric field DueDate to a "real" date

```
 * Determine if payment is more than 90 days overdue - prior to V5R1

C     *MDY0         MOVE      DueDate       WorkDate
C                   TIME                    Today

C     Today         SubDur    WorkDate      Temp:*D
C                   If        Temp > 90
C                   : ......  : ..............................
C                   EndIf

  // And the same calculation in /Free
   If %Diff( %Date(): %Date(DueDate: *MDY0): *Days) > 90;
      ..............:.............
   EndIf;
```

Returns the system date

In this example, you can see how much more powerful it is to be able to use date operations in expressions. In this small example alone, we did away with 3 temporary work fields. We think the new operation is also more obvious as to its purpose.

Note the use of %Date for 2 functions - first to retrieve the current date for the calculation and second to convert the DueDate field to a date form so it can be used in the duration calculation (via %DIFF).

Replace Call - PRrototype

Partner400

The modern way to call anything

- e.g., program or procedure or subprocedure

Parameter mismatches can become a thing of the past!

- The compiler can validate your program calls
- Prototypes allow the compiler to validate:
  - The number of parameters and
  - Their data type and size

Prototypes can accommodate differences in data definition

- For example, allowing an integer to be passed when the callee expects a value with decimal places

The op-code CALLP is used with Prototypes

- It means **CALL** using **P**rototype (and NOT Call Procedure)

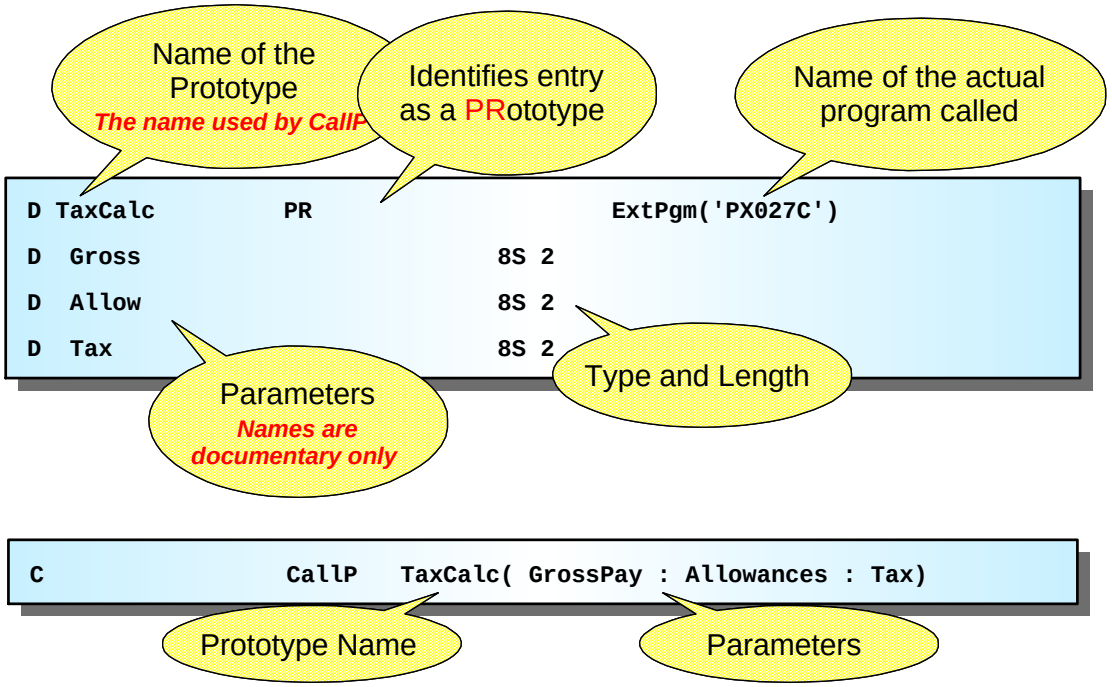
Prototypes can be used when calling anything - a program, whether RPGLE or RPG or CLP or CLLE, etc. or a subprocedure or a C function, a system API - anything.

Prototypes were added to the RPG IV language in the V3R2 and V3R6 releases. Although their initial purpose was to support Subprocedures, the RPG developers realized that they could be used to provide additional support for program calls among other things. This helps to address one of the really annoying problems with the old CALL/PARM syntax. Namely that errors with parameter lists are not discovered until run-time. It would be much better if we could have the compiler validate the parameter lists for us, and that's what prototyping provides.

In addition to validating parameters, prototyped calls allow for automatic adjustment in the case of certain parameter mismatches. For example they can allow you to pass a 7 digit packed field when a 9 digit zoned is expected. The techniques involved are beyond the scope of this session but you can find articles on our web site www.Partner400.com

A great many people mistakenly believe that prototypes and CALLP relate only to procedures. This is not true. CALLP can invoke either a program or a procedure. As we shall see shortly, which type of call is made depends on keywords on the PR line of the prototype.

Anatomy of a Prototyped Call



There are two main parts to a prototype

The first line is the PR line itself
- This supplies the name of the program or procedure to be called
- It also marks the beginning of the parameter list
- The PR entry goes in positions 24 -25 (the same place you would put DS for a data structure)

The second and subsequent lines describe the parameters in sequence
- Parameters are identified by a blank in positions 24-25 just as the subfields of a data structure are.
- The parameter list is terminated by the appearance of any non-blank entry
  - For example a DS, S, C, PI (more on this later) or another PR
- If the program or procedure to be called does not require any parameters - you don't define any!

Note that the names used for the parameters do not match the names of the fields in the CALLP
- In fact they could be completely blank and the compiler would be quite happy
- It only cares about the number and type of parameters - the names used are irrelevant
- Some people use a standard whereby the name used in the prototype identifies the type of field
  - e.g. Currency, Integer, Name, etc.

The CALLP itself is a free-form operation. The parameters are enclosed in parentheses and separated by colons.

If there are no parameters, then the parentheses are omitted. In V5R1 and later releases, an empty set of parentheses can be used instead. This is the preferred method as it makes the intent of the code more obvious and is compulsory in the /Free version of free-form RPG available in V5R1.

Simple Prototype Example

If we are using these field definitions:

```
D GrossPay                      8S 2
D Allowances                    6S 2
D Tax                           8S 2
```

Then the following CALL sequence:

```
C               Call      'PX027C'
C               Parm                    GrossPay
C               Parm                    Allowances
C               Parm                    Tax
```

Can be replaced by this Prototype and CALLP

```
D TaxCalc         PR                    ExtPgm('PX027C')
D  Gross                           8S 2
D  Allow                           6S 2
D  Tax                             8S 2

 /Free
   CallP TaxCalc( GrossPay : Allowances : Tax);
```

In our example the prototype has been hard coded in the calling program.
Normally we would expect to see it being /COPY'd in from a source file supplied by the programmer who wrote PX027C. This is good practice since who knows better what parameters the program is expecting?

This approach is going to seem like more work at first glance. It is! But there are two things it is important to remember:
First - you only have to code it once.
Second - our aim is not to speed the writing of the code, rather it is to reduce the opportunity for errors to be introduced during maintenance. Since 80% of the effort expended on most programs is in maintenance/enhancements, the extra time pales into insignificance.

Any variance from these rules will be noted and immediately rejected by the compiler. Compare this to a conventional CALL/PARM situation where the compiler would not produce an error, and the brand new bug could well be introduced into production.

Replacing the *ENTRY PLIST

A Procedure Interface (PI) can replace the *ENTRY PLIST

- But the Prototype must also be present
- Placing it in a /COPY member is the best idea

```
D TaxCalc         PR                  ExtPgm('PX027C')
D                                8S 2
D                                6S 2
D                                8S 2
```

Source TAXCALCPR

Note that the PI entries actually define the fields

- In the PR they simply provide information for the compiler

```
 /Copy TaxCalcPr
D TaxCalc         PI
D  GrossPay                      8S 2
D  Allowances                    6S 2
D  Tax                           8S 2
```

We can replace the *ENTRY PLIST by placing a Procedure Interface and Prototype in the called program.

Don't be tempted to simply clone the prototype using CC in SEU - do the job right and create a source member to contain the prototype and /COPY it into the called program and also use it in any program that calls it.

The added benefit for all your "hard work" is that the compiler can now validate the parameters to make sure they match the prototype. You can also designate fields as read-only via the keyword CONST. But all of those kind of capabilities require a session of their own.

Replacing The LOOKUP Op-Code

Partner400

One more nail in the *INnn Coffin!

%LOOKUPxx(SearchFor : Array { : StartIndex {: # Elems} })

- Provides the same basic function as the LOOKUP Op-Code
  - Unlike LOOKUP, SearchFor and Array do not have to match in size and decimal positions
  - Also provides optional number of elements to search in addition to starting index

%LOOKUP - Searches for an exact match

- %LOOKUPLE - Searches for an exact match, or the closest item lower
- There are also %LOOKUPLT, %LOOKUPGE, and %LOOKUPGT
  - I'll leave you to guess what they do!

Returns the Index number if SearchFor is found

- Otherwise it returns Zero
  - The LOOKUP Op-code would have set the index (if supplied) to 1
  - Also %LOOKUP will not change the value of the starting index

Note the differences in behavior between %LOOKUP and the corresponding operation code as shown on the bottom of this chart. Also note that the BIFs %FOUND and %EQUAL are NOT set as a result of using %LOOKUP, which is different for the operation code. Also note that %LOOKUP can NOT be used to look up values in tables (unlike the LOOKUP operation code)

However, those of you who use Tables will be pleased to know that there are corresponding table lookup versions of these BIFs. Not surprisingly they all start with %TLOOKUPxx.

The basic syntax for these BIFs is

```
%TLOOKUPxx(searchfor : table {: alttable {: # elems} } )
```

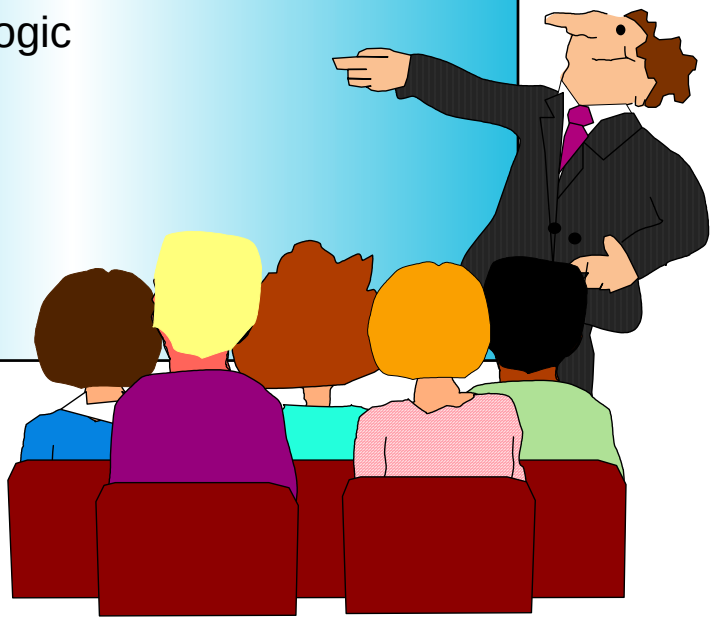
%TLOOKUPxx sets the current table element for the specified table and also the alternate table if specified.

In case the meaning of "One more nail in the *INnn Coffin!" escapes you - LOOKUP was one of a very few operation codes as of V4R4 that still required the use of resulting indicators (the Hi/Lo/Eq type) and so it required that numbered indicators still be used. Now, programmers can (and should!) use the %LOOKUP BIF instead and remove some of those last remaining numbered indicators - replacing them instead with either named indicators (e.g., to communicate with Display Files) or BIFs, such as %EOF, %FOUND, %ERROR, etc.

Now Appearing in /Free Only

Partner400

New options only available
in /Free format logic



I/O Enhancements - New Key Options

Partner400

%KDS is a free-form replacement for KLIST

- Used in Factor 1 position instead of a KLIST name

Syntax - %KDS(keyDSName { : numberOfKeysToUse })

- Optional second argument specifies number of keys to be used

All keyed operations can use this new BIF

Generate related DS by using LIKEREC(recname : *KEY)

```
   // The ProductKeys DS will contain the three fields that make up the
   //   key for the Product File (Division, PartNumber and Version)
D ProductKeys     DS                  LikeRec(ProductRec : *Key)

 /Free
 // Read specific record using full key
   Chain %KDS(ProductKeys) ProductRec;

 // Position file based on first 2 key fields
   SetLL %KDS(ProductKeys : 2 ) ProductRec;
```

Note V5R2:
Supported only in /Free

/Free ONLY

Until now, there were two ways to specify the key for a keyed operation. Specify the name of a single field or the name of a KLIST. KLISTs always annoyed me because you had to wander off elsewhere in the program to actually find the list. Only then did you know what keys were being used. This new support offers both an improved alternative to the KLIST approach and a new method of directly specifying the keys on the operation itself.

The new "KLIST" (actually a BIF called %KDS - Key Data Structure) references key definitions in the D specs where they belong. Remember the LIKEREC *KEY option we covered earlier? This is where it comes into play. You can use it to automatically generate a DS containing the file's key fields. This structure can then be referenced in the I/O operation by specifying the DS name to the new %KDS function.

So how do you specify that a partial key is to be used? Just use the second parameter of %KDS to tell the compiler how many of the key fields are to be used.

We will look at the second method on the next chart.

I/O Enhancements - New Key Options

Keys can now also be specified as a list of values

- List is specified in Factor 1 position, enclosed in parentheses

Any type of field, literal or expression can be used

- As long as the base type matches
  - i.e. The result is numeric for a numeric key, Alpha for an alpha key, etc.
- The compiler will perform any required conversions
  - Using the same rules as for EVAL operations

Note V5R2:
Supported only in /Free

```
/Free
// Read using specified keys
  Chain (Division : PartNumber : Version) ProductRec;

// Position file based on first 2 key fields
  SetLL (Division : PartNumber) ProductRec;

// Position file to specified Part number in Division 99
  SetLL ( '99' : PartNumber) ProductRec;
```

/Free ONLY

The second method is an extension of the current ability to specify a single field as the key (the old Factor 1).

Instead of a single field, you can now supply a list of fields. The list must be specified within parentheses with colons (:) used to separate the individual key elements.

Note that the key elements do not have to be fields, they can be any character expression. The compiler will perform conversion if required.

I/O Enhancements - %FIELDS

Allows you to specify the list of fields affected by an UPDATE

- Only those fields specified will be updated

Syntax - %FIELDS(name1 { : namen ... })

Used in the Result position of the op-code

Note V5R2: Supported only in /Free

```
FProduct   UF   E           K DISK
D ProductData     DS                  LikeRec(ProdRec:*ALL)
 /free
  // Update record using values in ProductData DS
   Update Product ProductData;
  // Update only UnitCost and UnitPrice fields
   Update Product %Fields( UnitCost : UnitPrice );
```

/Free ONLY

In many ways this is one of the best of the new features, and we saved it for the end (almost)

This is the capability to limit which fields are modified by an UPDATE operation. We love this one! It provides a great way to protect your code from the worst efforts of (shall we say) less-gifted programmers. The list of fields is specified using the new BIF %Fields. **Only** those fields specified will be updated.

Why is this useful? Suppose that, during the operation of the program, only certain fields in the file should be subject to change. By specifying those fields to the UPDATE op code, you are assured that only those fields will be changed. If during subsequent maintenance tasks a mistake is made and the value of a field that should not be modified is accidentally changed in the code, it will have no effect on the database. Only if the %Fields list is also modified can this error result in database corruption.

Shortform Expression Support (V5R2)

We suspect you'll either love this or hate it

- It is kinda nice when using subscripted or qualified names
  - Particularly if you are a slow typist !!

It allows you to shorten expressions of the type X = X + 1

- Instead of repeating the field name (X) you can use shortforms
  - += for addition
  - -= for subtraction
  - and you can guess what the others are ...

This is not limited to /Free

```
 // This calculation
    MonthTotal(Month) = MonthTotal(Month) + TotalSale;
 can be shortened to this
    MonthTotal(Month) += TotalSale;
// and this
    InvoiceTotal = InvoiceTotal - CustDiscount;
// can similarly be shortened to this
    InvoiceTotal -= CustDiscount;
```

Numeric operations now support short-form notation for certain functions. Prior to this release, an addition of the type **X = X + 1** required that you repeat the name of the target field. Some people considered this a step backwards since the old ADD op-code offered a short-form notation that only required the target field to be specified once, in the result field. e.g. **ADD 1 X**.

With this new feature, the expression can be written as **X += 1**. Similar shorthand can be used for subtraction, multiplication, division, and exponentiation.

I find the syntax a little confusing because my first thought is to key it as **X =+ 1** and in fact I have made that mistake several times already. I will learn in time!

Resources

Linoma Software's RPG Toolbox

- The latest evolution of the first RPG III conversion utility
  - Now accommodates /Free formatting and much more
- www.LinomaSoftware.com and follow the RPG Toolbox links

Articles by Jon Paris and Susan Gantner

- Link to them from the Partner400 web site
  - Partner400.com/Articles.htm

Book: Free-Format RPG IV: How to Bring Your RPG Programs Into the 21st Century

- by Jim Martin
- Available from MC Press