THE ESSENTIAL GUIDE TO

# Free-Format RPG

BY BRYAN MEYERS — MAY 2005

IN THE INCREASINGLY COMPLEX WORLD of business programming, RPG continues to have a strong role. Despite the comings and goings of new technologies, billions of lines of legacy RPG code still provide the steady backbone for many applications, running businesses both large and small.

But the character of RPG code has evolved over recent years, adapting to interoperate with new languages, file systems, user interfaces, and program construction models. Arguably, the two most important evolutionary leaps occurred at Version 3 with the introduction of the RPG IV syntax itself, and at Version 5, which effectively replaced C-specs with free-format code.

To help you transition from fixed-format RPG IV coding to free format this guide will cover free-format mechanics, best practices, and conversion tips. (See "The Essential Rationale" for 10 reasons you should favor free-format RPG.)

## A Free-Format Primer

The free-format specification (Figure 1) is basically an alternative to traditional columnar C-specs. You use familiar RPG syntax to code the free-format specification between /Free and /End-free compiler directives, but you don't have to concern yourself with getting everything in the right columns. You can specify code anywhere from columns 8 to 80, leaving columns 6 and 7 blank.

The actual content of the free-format code is familiar except that each line starts with an operation code instead of Factor 1. The correct order for a specification is

```
Opcode(e)   Factor1   Factor2   Result;
```

Figure 1. Free-format RPG snippet

```
    // ****************************************************************
    //
    // Main processing
    //
   /Free
    Dou *Inlr;
      Read(e) Mmaster1;

      If %Error Or %Eof;
        *Inlr = *On;
        Leave;
      Endif;

      Count = Count + 1;                                  // Implied Eval
      Expireday = Dayname(Expiredate);
      Buildlabel();                                       // Implied Callp
      Printrec();

    Enddo;

    If Overflow;
      Write(e) Header;
    Endif;

    Write Endjob;
    Return;
   /End-free

    //****************************************************************
    //
    // Procedure - Dayname
    //
  P Dayname B
    // ------------------------------------------------ Procedure interface
  D                 PI             9
  D Date                            D

    // ------------------------------------------------------ Local variables
  D DayNbr          S              10U 0


   /Free
    Daynbr = %Diff(Date:D'1899-12-30':*Days);
    Daynbr = %Rem(Daynbr:7);

    If Daynbr > 0;
      Return Days(Daynbr);
    Else;
      Return Days(Daynbr) + 7;
    Endif;
   /End-free
  P DayName E
```



SUPPLEMENT TO iSeries NEWS © 2005

Factor 1, if there is one, follows the operation code. Factor 2 and the Result then trail Factor 1. At least one space must separate the pieces of the statement; however, the operation code extender must be joined to the operation code with no space.

Free-format RPG uses many of the same operation codes as traditional fixed-format RPG IV — there are no new operation codes to learn, and in two cases, the operation code is optional. You can omit EVAL (Evaluate expression) and CALLP (Call prototyped procedure/program) unless you need to specify an extender, as the following example shows:

```
Tax = TaxableAmount * TaxRate;
Eval(h) Tax = TaxableAmount * TaxRate;

UpdCustomer(Company:CustomerNbr);
Callp(e) UpdCustomer(Company:CustomerNbr);
```

You can align and/or indent code to make it more readable and continue a specification on to several lines if necessary (but only one statement is allowed on a line). Each free-format statement terminates with a semicolon delimiter (see "An Essential Debugging Tip" to learn about missing semicolon delimiters).

Comments in free-format code begin with two slashes (//) instead of an asterisk (*) in column 7. The comment can begin anywhere from columns 8 to 80, and you can even code comments "inline" with a specification (see "The Essential Rules" for a summary of free-format coding points). Double-slash comments may also appear instead of asterisk comments in fixed-format RPG specifications starting anywhere from columns 6 to 80, but they must be alone on a line without any other specification code. Free-format code may also include two compiler directives, /Copy and /Include. The compiler considers /Copy members as fixed-format code, even if the /Copy directive is in a free-format block. Conditional compiler directives — /Define, /Undefine, /If, /Else, /Elseif, /Endif, /Eof — can also appear in a free-format block and can be indented (contrary to the documentation in the RPG IV reference manuals).

## Indicators Begone!

A striking feature of free-format RPG is its elimination of indicator columns. Conditioning indicators, resulting indicators, and control level indicators are not allowed; the space formerly reserved for those indicators is now open for real work.

Instead of conditioning indicators (columns 9–11), you should test for conditions using the structured operation codes (e.g., If, Dou, Dow, For, Select). RPG IV offers numerous built-in functions (BIFs) to replace resulting indicators. The most common ones are %Found, %Error, and %Eof for the file I/O operation codes. (Figure 2 shows you which functions are valid with which operation codes.) The %Lookup and %Tlookup functions replace the Lookup operation code and perform array and table lookup functions without resulting indicators.

**Figure 2 File I/O functions**

| Operation | %Found | %Error | %Eof | %Equal |
|---|---|---|---|---|
| Chain | Valid | Valid | — | — |
| Delete | Valid | Valid | — | — |
| Read | — | Valid | Valid | — |
| Readc | — | Valid | Valid | — |
| Reade | — | Valid | Valid | — |
| Readp | — | Valid | Valid | — |
| Readpe | — | Valid | Valid | — |
| Operation | Valid | — | — | — |
| Operation | Valid | Valid | — | Valid |
| Operation | — | Valid | Valid* | — |

**Valid only when writing a subfile record*

Any operation code that uses an error resulting indicator (in columns 73 and 74) can be coded with an (E) error-handling extender instead. If an error occurs during the operation, %Error will be set to *On, and %Status will return a status code to identify the error.

Finally, if you're still using level indicators from the RPG cycle, those indicators (columns 7 and 8) can't appear in free-format code. If you must use them, you'll need to momentarily fall out of free-format syntax, enter the level indicator, and then resume free format:

```
/Free
   (Free format code block)

/End-free
L2
/Free
 Write(e) L2Totals;
   (This code executes at an L2 break)

/End-free
L1
/Free
 Write(e) L1Totals;
   (This code executes at an L1 break)
```

Best practices dictate that you avoid using conditioning indicators and level indicators — free format enforces that standard to help clean up RPG.

## A Lean, Clean Programming Machine

Another way that free-format RPG enforces best practices is by consolidating all of its functions into about 60 operation codes — half the number that fixed-format RPG supports. (For a list of all the operation codes that you can include in a free-format block, see "The Essential Syntax."

As for the operation codes that free format left behind, you should consider them obsolete. Most of them have better, more modern alternatives. For example, the arithmetic operations (Add, Sub, Mult, Div, and so on) can substitute familiar operators in free-format expressions. Alternatively, simple assignment expressions such as Z-add or Z-sub can replace arithmetic operation codes.

BIFs replace many of the jettisoned operations; they

provide more flexible coding, greater function, and more enhancement potential than their corresponding operation codes. Frequently, you can simply substitute a function for an operation code, including the function in an expression. ("The Essential Alternatives" summarizes the alternatives for the unsupported operation codes.)

## The MOVE to Free Format

The Move operations (Move, Movel, and Movea) were relatively powerful, handling a wide variety of data types and sizes with specific rules for each circumstance. Sometimes, a Move operation simply assigned the value of one data item or literal to another variable; other times, it may have performed data type conversions, such as converting a number to a date or a character field to a number. If the Move target was a different size than the source value, other special rules applied, depending upon which Move operation you were using.

These operation codes have several free-format alternatives, and therein lies the challenge for many programmers. When you're writing a new program, it's easy to avoid using Move, but converting a legacy program to use the alternatives isn't so simple. Choosing the correct alternative for any given Move scenario requires an examination of three factors: data types, field sizes, and the desired results.

Predictably, the easiest conversion occurs when the Move source and target are both the same data type and length. A simple assignment expression will do the job:

```
Target = Source;
```

To go beyond this simple assignment, however, you must usually employ any of several BIFs in an assignment expression.

When both the source and target are character data, Eval is equivalent to the Movel(p) (Move left with blank padding) operation. Evalr (Evaluate with right adjust) is equivalent to Move(p):

```
Target = Source;           // MOVEL(P)
Evalr Target = Source;     // MOVE(P)
```

To avoid padding the result with blanks, consider the lengths of the respective fields. If the result is shorter than the source, use Evalr instead of Move to assign the source to the target:

```
Evalr Target = Source;
```

If the target variable is longer than the source, you can use %Subst on the left side of the assignment expression, thereby assigning the source to only a portion of the result:

```
%Subst(Target:
     %Size(Target)-%Size(Source)+1) = Source;
```

Usually, you would simplify this code. Let's say the source is five bytes and the target is nine bytes:

```
%Subst(Target:5) = Source;
```

To replace a Movel operation without padding, you need to adjust the locations, but you still use %Subst if the target variable is longer than the source:

```
%Subst(Target:1:%Size(Source)) = Source;
```

Or, for the earlier scenario:

```
%Subst(Target:1:5) = Source;
```

Of course, if the target is shorter than the source, a simple assignment will do the trick in a Movel conversion:

```
Target = Source;
```

## Converting Data Types in Free Format

The Move-related operations are not restricted to "same type" moves. You can use Move to transfer a numeric value to a character string and back again. The Move operations are also aware of the native date/time data types, letting you easily move into and out of dates with few restrictions or special coding requirements. To accomplish this feat in free format requires the use of one or more data conversion functions. Figure 3 illustrates possible alternatives for various data types.

**Figure 3 Alternatives to Move**

| Move Factor 1 Data Type | Move Results Data Type | Alternative Contruct(s) |
|---|---|---|
| Character | Character | EVAL or EVALR, with or without %SUBST function |
| Numeric | Character | %EDITC function with X edit code, with or without %SUBST function |
| Date | | Character %CHAR function |
| Character | Numeric | %DEC, %INT, %UNS functions (V5R2) |
| Numeric | Numeric | Simple EVAL |
| Date | Numeric | %DEC, %INT, %UNS functions, with %CHAR function (V5R2) |
| Character | Date | %DATE, %TIME, %TIMESTAMP functions |
| Numeric | Date | %DATE, %TIME, %TIMESTAMP functions |
| Date | Date | Simple EVAL |

You can use %Editc to assign a numeric value to a character string. To replace a fixed-format Move operation, use the X edit code to return a simple numeric source value as a character value:

```
Target = %Editc(Source:'X');
```

If the source and result are sized differently, you'd use %Subst in addition to %Editc. The X edit code works equally well for packed and zoned (signed) numbers and for positive and negative numbers.

The other data type conversions in Figure 3 are relatively straightforward (using one of the data conversion functions in an assignment expression) and don't require additional explanation.

## Moving Arrays in Free Format

The Movea (Move Array) operation performs some specialized assignment functions related to arrays. It will move several contiguous array elements to a variable, a variable to several contiguous array elements, and contiguous elements between two arrays.

As with the other Move operations, Movea is not valid in free-format code. In V5R3, though, the %Subarr (Get/Set Portion of an Array) function can appear in assignment expressions to perform the same functions as Movea. Using %Subarr lets you refer to a subsection of an array, starting at a given index and continuing for a given number of elements:

```
%Subarr(Array:Start:Elements)
```

The first two required arguments name the array and the starting index. The third argument tells %Subarr how many elements you want to process; if you omit it, you will process the remainder of the array.

To assign several contiguous array elements to a variable, you can use %Subarr to designate which elements you want to move:

```
Target = %Subarr(Array:Start:Elements);
```

On the left side of an assignment expression, %Subarr will change contiguous elements in an array. To move a source variable to contiguous elements of an array, you can use a simple assignment expression:

```
%Subarr(Array:Start:Elements) = Source;
```

To move contiguous elements between two arrays, use %Subarr on both sides of the expression:

```
%Subarr(Target:Start:Elements) =
              %Subarr(Source:Start:Elements);
```

If the array elements differ in length or data type, use other functions in conjunction with %Subarr to get the results you want.

## Fixed Format and Free Format Don't Mix Well

As noted earlier, if you must revert to fixed-format code for a line or two in the middle of a free-format code block, you can temporarily drop out of free-format syntax with the /End-free directive. But avoid mixing traditional style and free-form style in your C-specs. The result is inconsistent and difficult to read.

However, don't completely abandon columnar alignment as a tool to aid readability in expressions, particularly when an expression must continue on to subsequent lines. The following example shows how aligning an expression can make it easier to understand:

```
TotalPay = (RegHours * Rate)       +
           (OvtHours * Rate * 1.5) +
           (DblHours * Rate * 2);
```

Currently, the free-format specification replaces only C-specs, not the other specification types. Before you can code a procedure, for example, you'll need to code an /End-free directive before the beginning P-spec, then include a /Free directive after any D-specs before you code free format again, and finally code /End-free again before the ending P-spec (for a wish list of further free-format enhancements, see "The Missing Essentials"). Unfortunately, embedded SQL statements still require a fixed-format entry. Until IBM corrects this glaring design gaffe, you can alleviate the eyesore somewhat by indenting the RPG code to match the SQL code, as in Figure 4.

**FIGURE 4**
Aligning embedded SQL statements

```
           PgmAccount = 'G5X67';
 /End-free
C/Exec SQL Select FirstName, LastName, City, State
C+           Into :InFirstName, :InLastName,
C+                :InCity, :InState
C+           From MMaster
C+           Where Account = :PgmAccount
C/End-exec
 /Free
           If InState = 'TX';
```

**Bryan Meyers** is a technical editor for *iSeries NEWS* and author of several books, including *RPG IV Jump Start* and *Programming in RPG IV* (both from 29th Street Press). Bryan presents classes on several iSeries topics — on site, on DVD, or through the iSeries Network's e-Learning program. You can reach him at bmeyers.net.

## THE ESSENTIAL RATIONALE

**Here are the top 10 reasons you should favor free-format RPG over fixed-format code:**

10. It's compatible. You can mix free-format and fixed-format code. Free-format code supports a subset of the same operation codes used by fixed format.
9. It's bigger. Free format offers more room for longer expressions.
8. It's lucid. Inline comments are better documentation than right-hand comments. Because you can position inline comments, they are more documentary and relevant.
7. It's organized. You can indent nested logic for easier comprehension.
6. It's logical. Free-format code is easier to read. If you start the specification with an operation code, free-format RPG programs become more action oriented.
5. It's faster. Free-format code is faster to enter. If you're using SEU as your editor, you can eliminate the horrific prompter function and "just code it."
4. It's familiar. Free-format RPG source closely resembles other widely understood languages. If you're primarily an RPG programmer, you'll find less of a learning curve when you deal with C or Java.
3. It's easier. Free-format code is easier to teach to new programmers. If you're hiring programmers, they'll have an easier time learning RPG because it will already look familiar.
2. It's fresh. Free-format code eliminates obsolete language components, and it enforces many best practices by simply rejecting old operation codes and "worst practices."

**And the number one reason you should favor free-format RPG:**

1. It's the future. Most new RPG enhancements are supported only in free format. Now that the syntax need not fit into fixed-length factors, IBM has much more flexibility in adding capabilities to the language.

## THE ESSENTIAL RULES

1. Free-format code appears between /Free and /End-free directives.
2. Code belongs in columns 8 to 80; columns 6 and 7 must be blank.
3. Only one statement can be coded on a single line.
4. Each statement must end with a semicolon (;) delimiter.
5. Each statement begins with an operation code.
6. Comments must begin with double slashes (//).
7. Free-format code prohibits the use of any indicator columns.
8. Free-format code prohibits obsolete operation codes.

# THE MISSING ESSENTIALS

Especially with V5R3, there's really no valid rationale for staying fixed in the past. With free-format RPG, you can completely eliminate fixed-format C-specs. But that doesn't mean there's no room for further enhancement. Here's a wish list:

- Support free-format specifications with embedded SQL. The current implementation, which requires you to revert to 1960s-style code for embedded SQL statements, is hideous.
- Eliminate the need for the /Free and /End-free directives. The compiler should be smart enough to recognize fixed-format code as anything with an entry in column 6.
- Free the other specifications. Especially the commonly used H-, F-, D-, and P-specs should be replaced by free-format identifiers (e.g., Control, File, Define, Procedure).

## AN ESSENTIAL DEBUGGING TIP

The most common error you'll likely make when coding free format is to forget the semicolon delimiter at the end of each line. Forgetting the semicolon will result in compiler errors — and sometimes the error message won't tell you that you forgot the semicolon. If I'm having difficulty understanding a compiler error message in a free-format program, I look to the source line just above the source line that generated the error. Often, the error will simply be a missing semicolon that managed to confuse the compiler.

## THE ESSENTIAL ALTERNATIVES

| | |
|---|---|
| ADD | Use + operator |
| ADDDUR | Use + operator, date functions |
| ALLOC | Use %ALLOC |
| ANDxx | Use AND |
| BITOFF | Use %BITAND, %BITNOT (V5R2) |
| BITON | Use %BITOR (V5R2) |
| CABxx | Use logical expressions |
| CALL | Use CALLP |
| CALLB | Use CALLP |
| CASxx | Use SELECT/WHEN/OTHER |
| CAT | Use + operator |
| CHECK | Use %CHECK |
| CHECKR | Use %CHECKR |
| COMP | Use logical expressions |
| DEFINE | Use D-specs with LIKE |
| DIV | Use / operator or %DIV |
| DO | Use FOR |
| DOUxx | Use DOU |
| DOWxx | Use DOW |
| EXTRCT | Use %SUBDT |
| GOTO | No alternative |
| Ifxx | Use IF |
| KFLD | Use %KDS data structure (V5R2) |
| KLIST | Use %KDS data structure (V5R2) |
| LOOKUP | Use %LOOKUP or %TLOOKUP |
| MHHZO | Use %BITAND, %BITOR (V5R2) |
| MHLZO | Use %BITAND, %BITOR (V5R2) |
| MLHZO | Use %BITAND, %BITOR (V5R2) |
| MLLZO | Use %BITAND, %BITOR (V5R2) |
| MOVE | Use EVALR, %SUBST, conversion functions |
| MOVEA | Use %SUBST, %SUBARR (V5R3), conversion functions |
| MOVEL | Use %SUBST, conversion functions |
| MULT | Use * operator |
| MVR | Use %REM |
| OCCUR | Use array data structure (V5R2) |
| Orxx | Use OR |
| PARM | Use PR/PI definitions |
| PLIST | Use PR/PI definitions |
| REALLOC | Use %REALLOC |
| SCAN | Use %SCAN |
| SETOFF | Use assignment expressions |
| SETON | Use assignment expressions |
| SHTDN | Use %SHTDN |
| SQRT | Use %SQRT |
| SUB | Use – operator |
| SUBDUR | Use – operator, %DIFF, date functions |
| SUBST | Use %SUBST |
| TAG | No alternative |
| TESTB | Use %BITAND (V5R2) |
| TESTN | Use MONITOR, ON-ERROR |
| TESTZ | Use %BITAND (V5R2) |
| TIME | Use %DATE, %TIME, %TIMESTAMP |
| WHENxx | Use WHEN |
| XFOOT | Use %XFOOT |
| XLATE | Use %XLATE |
| Z-ADD | Use EVAL |
| Z-SUB | Use EVAL |



SUPPLEMENT TO iSeries NEWS © 2005

## THE ESSENTIAL SYNTAX

(Required entries are colored red.)

```
ACQ(e)  device-name  workstn-file ;
BEGSR  subroutine-name ;
CALLP(emr)  name(parm1:parm2...) ;
CHAIN(enhmr)  search-arg  name  data-structure ;
CLEAR  *NOKEY  *ALL  name ;
CLOSE(e)  file-name ;
COMMIT(e)  boundary ;
DEALLOC(en)  pointer-name ;
DELETE(ehmr)  search-arg  name ;
DOU(mr)  logical-expression ;
DOW(mr)  logical-expression ;
DSPLY(e)  message  output-queue  response ;
DUMP(a)  identifier ;
ELSE ;
ELSEIF(mr)  logical-expression ;
ENDDO ;
ENDFOR ;
ENDIF ;
ENDMON ;
ENDSL ;
ENDSR  return-point ;
EVAL(hmr)  assignment-expression ;
EVALR(mr)  assignment-expression ;
EXCEPT  except-name ;
EXFMT(e)  format-name ;
EXSR  subroutine-name ;
FEOD(en)  file-name ;
FOR(mr)  index = start  BY increment  TO|DOWNTO limit ;
FORCE  file-name ;
IF(mr)  logical-expression ;
IN(e)  *LOCK  data-area-name ;
ITER ;
LEAVE ;
LEAVESR ;
MONITOR ;
NEXT(e)  program-device  file-name ;
ON-ERROR  exception-id1:exception-id2... ;
OPEN(e)  file-name ;
OTHER ;
OUT(e)  *LOCK  data-area-name ;
POST(e)  program-device  file-name ;
READ(en)  name  data-structure ;
READC(e)  record-name  data-structure ;
READE(enhmr)  search-arg  name  data-structure ;
READP(en)  name  data-structure ;
READPE(enhmr)  search-arg  name  data-structure ;
REL(e)  program-device  file-name ;
RESET(e)  *NOKEY  *ALL  name ;
RETURN(hmr)  expression ;
ROLBK(e) ;
SELECT ;
SETGT(ehmr)  search-arg name ;
SETLL(ehmr)  search-arg  name ;
SORTA  %Subarr(array-name:start:number-of-elements) ;
TEST(edtz)  dtz-format  field-name ;
UNLOCK(e)  name ;
UPDATE(e)  name  data-structure|%FIELDS(name1:name2...) ;
WHEN(mr)  logical-expression ;
WRITE(e)  name  data-structure ;
```



SUPPLEMENT TO iSeries NEWS © 2005