A P P E N D I X D

This appendix is from the textbook "Programming in ILE RPG, 5th ed." Due to space constraints, this material was not included in the printed book.

©Copyright 2015, Jim Buck and Bryan Meyers. All rights reserved.

ILE RPG Style Guide

OVERVIEW

This appendix presents some suggestions for how to write RPG programs that are easy to read, understand, and maintain. Professional programmers appreciate the importance of standards in developing programs. The issue of program style goes beyond any one language, but the ILE RPG syntax has some unique features that promote specific style guidelines. Here are some simple rules that you can use to ensure that bad code doesn't happen to otherwise good RPG software construction.

DEFINING DATA

Within the Declarations section of the program, organize your definitions in a predictable order by definition type:

- File declarations
- Named constants
- Prototype for main procedure
- Procedure interface for main procedure
- Other prototype definitions
- Data structures
- Standalone variables

Within each definition type, alphabetize the declarations.

Expand Naming Conventions

Perhaps the most important aspect of programming style deals with the names you give to data items (e.g., variables, named constants) and routines. Establish naming conventions that fully identify variables and other identifiers. A good naming strategy can make the difference between program *code* and a program *description*.

When you name an item, be sure the name fully and accurately describes the item. The name should be unambiguous, easy to read, and obvious. Although you should exploit RPG's allowance for long names, don't make your names too long. Name lengths of 10 to 14 characters are usually sufficient, and longer names may not be practical in many specifications.

A data item's name should describe the item. When naming a procedure with a return value, name the procedure after the return value, or use a get/set naming convention if the procedure retrieves or assigns a data value.

For subroutines or procedures without a return value, use a verb/object syntax (similar to a CL command) to describe the process. Maintain a dictionary of names, verbs, and objects, and use the dictionary to standardize your naming conventions.

Avoid using special characters (e.g., @, #, $) when naming items. Some of these characters can cause compile errors in some character sets. Although ILE RPG allows an underscore (_) within a name, you can easily avoid this *noise* character if you use mixed case intelligently.

Declare Named Constants Instead of Using Literals

This practice helps document your code and makes it easier to maintain. One obvious exception to this rule is the allowable use of 0 and 1 when they make perfect sense in the context of a statement. For example, to initialize an accumulator field or increment a counter, you can use a hard-coded 0 or 1 in the source.

Indent Data Item Names

Use indentation to document data structures and their subfields:

```
Dcl-ds Errmsg;
  Errprefix   Char(3);
  Errmsgid   Char(4);
    Errmajor Char(2) Overlay(Errmsgid);
    Errminor Char(2) Overlay(Errmsgid:*Next);
End-ds;
```

Avoid Compile Time Arrays

As you decompose a program into its individual procedures, it helps to have all related pieces of code physically and logically self-contained. The traditional compile time array coding separates the array definition from the array data by perhaps thousands of code lines in the program. A better solution is to code the array (Days in the following example) inside a data structure (Daysctdata):

```
Dcl-ds Daysctdata;
  *N   Char(9) Inz('Sunday');
  *N   Char(9) Inz('Monday');
  *N   Char(9) Inz('Tuesday');
  *N   Char(9) Inz('Wednesday');
  *N   Char(9) Inz('Thursday');
  *N   Char(9) Inz('Friday');
  *N   Char(9) Inz('Saturday');
  Days Char(9) Dim(7) Pos(1);
End-ds;
```

Avoid Multiple-Occurrence Data Structures

The array data structure is a better construct because it uses standard array notation in your program. In addition, it doesn't limit you to processing a single occurrence in the same line, and it allows you to deal with those instances where nested (multiple dimension) arrays are useful.

```
Dcl-ds Customers Dim(100) Qualified;
  Name       Char(35);
  Address    Char(35) Dim(2);
  City       Char(21);
  State      Char(2);
  Postalcode Char(10);
End-ds;
```

Use Qualified Data Structures

Qualified data structures force you to qualify their subfield names with the data structure name (e.g., Customer.Name, Customer.Address). This feature not only provides clear documentation for the data item, indicating where a data item originates, but it also allows you to have identically named subfields in different data structures (e.g., Before.Name, After.Name). To define a qualified data structure, use the Qualified keyword in its definition.

The Likeds and Likerec keywords are also useful when a data structure *inherits* its subfields from another data structure or record format. Data structures using Likeds and Likerec are implicitly qualified, without coding the Qualified keyword. You can control the subfields that appear in a Likerec data structure or an externally described data structure by specifying *All, *Input, *Ouput, or *Key fields.

FREE-FORMAT SYNTAX

At current releases, ILE RPG is a free-format language. Free-format specifications effectively render the traditional fixed-format specifications obsolete. When maintaining older programs, you will undoubtedly encounter fix-format code. The advantages of free format are well proven. It is easier to read, document, and maintain than fixed-format code, and its syntax is consistent with other modern computer languages. You should use it exclusively, especially when writing new programs or performing extensive maintenance on existing programs. In many cases, if you use the free-format specification, good standards will result automatically because the free-format specifications don't permit much of the obsolete *baggage* and poor practices that fixed-format specifications allowed.

Avoid mixing fixed-form style and free-form style in your programs. The result is inconsistent and difficult to read. Take full advantage of the more natural order and expanded space that the free-form specification affords.

Indent Code in Loops and Groups

When coding loops and groups, indent the code within a group by two spaces to highlight the structure of the code group, as the following examples illustrate:

```
If %Found;
  Eval(h) Totalpay = Regpay + Ovtpay;
Endif;

Select;
  When Hours <=40;
    Totalpay = Hours * Rate;
  When Dblhours = 0;
    Totalpay = (Hours * Rate) + ((Hours - 40) * Rate * 1.5);
  Other;
    Totalpay = (Hours * Rate) + ((Hours - 40) * Rate * 1.5) +
               (Dblhours * Rate * 2);
Endsl;

Dou %Eof(Master);
  Read(e) Master;
  Select;
    When %Eof(Master);
      Leave;
    When %Error;
      Exsr Errsubr;
    Other;
      Exsr Process;
  Endsl;
Enddo;
```

But don't completely abandon columnar alignment as a tool to aid readability in expressions. Especially when an expression must continue onto subsequent lines, align the expression to make it easier to understand:

```
Totalpay = (Reghours * Rate)       +
           (Ovthours * Rate * 1.5) +
           (Dblhours * Rate * 2);
```

If your program includes embedded SQL statements, integrate the SQL completely by using the same indenting standards for the SQL statements that you use for the rest of the program:

```
If Instate = 'TX';
  Pgmaccount = 'G5X67';
  Exec SQL Select Firstname, Lastname, City, State
           Into :Infirstname, :Inlastname, :Incity, :Instate
           From Master
           Where Account = :Pgmaccount;
  Exsr Process;
Endif;
```

For a long SQL statement, as in this example, consider coding the individual SQL predicates on separate lines and aligning them as you would multiline expressions.

Use Mixed-Case Source

Closely related to using free-format syntax is the standard to use mixed-case characters in the source code. Exploit this capability to make your program source easy to read. When coding a symbolic name, use mixed case to clarify the named item's meaning and use. Do not code in ALL UPPERCASE or all lowercase characters. Instead, use a logical combination of uppercase and lowercase, perhaps capitalizing each word in the code line:

```
Chain Postalcode Citymaster;
```

Another possibility is to use a notation commonly called *CamelCase*, which joins compound words without spaces, capitalizing each section of the compound:

```
Chain PostalCode CityMaster;
```

Only if your installation integrates Java programs into its applications, consider using Java's traditional Hungarian notation, a variation of CamelCase in which the variable's name indicates its type; the name begins with lowercase characters. Using Hungarian notation, the name unsCounter might indicate an unsigned integer, and decAmount might signify a packed decimal number.

For RPG-reserved words and operations, and possibly for named constants, consider using all uppercase characters. No matter which standard you use, be consistent.

MODULAR APPLICATIONS

The RPG syntax, along with the Integrated Language Environment (ILE), encourages a modular approach to application programming. Modularity offers a way to organize an application, facilitate program maintenance, hide complex logic, and efficiently reuse code wherever it applies.

Write Modular Programs with Procedures

Isolate reusable code in a procedure. Instead of monolithic, all-purpose monsters, write smaller, single-function compile units and bind them to the programs that need them. Eliminate duplicate code, even little bits of it. Your coding mantra should be "once and only once."

Relegate mysterious code to a well-documented, well-named procedure. Despite your best efforts, on extremely rare occasions, you simply will not be able to make the meaning of a chunk of code clear without extensive comments. Separating such heavily documented, well-tested code into a procedure will save future maintenance programmers the trouble of deciphering and dealing with the code unnecessarily.

Procedures and subroutines share many of the same advantages in building modular programs. But procedures have many features that subroutines do not support—parameter passing, local variables, Nomain modules, and service programs, for example. Your standards should favor using procedures over subroutines as a modular programming mechanism.

Use Return Values

As a general rule, a procedure should always return a value to its caller, even if that value is nothing more than an indicator of whether the procedure executed successfully. This syntax allows you to embed a procedure call inside an expression, and it makes the program easy to read:

```
Metrictemp = Celsius(Englishtemp);

If Isweekday(Shipdate);
  …
Endif;

If GetCustomer(Custnbr);
  …
Endif;
```

Use Binding Directories Consistently

A binding directory is an object that can help organize the pieces required to create a program. Using binding directories to list often-reused modules or service programs prevents tedium and errors by letting you refer to the binding directories instead of explicitly listing the required components when you bind a program. To use binding directories effectively, you need a consistent strategy for them. Perhaps the most useful strategy is to have a generic binding directory that refers to reusable code that crosses applications, along with an application-specific binding directory for code that relates only to one application.

Package Often-Reused Procedures in Service Programs

The service program is an elegant means of reusing procedures without physically copying them into each program that needs them. Generally, if a procedure is to appear in more than just one or two programs, package that procedure in a service program.

Use Binder Language to Control a Service Program Signature

Binder language source lets you explicitly control a service program's signature, and more important, allows the service program to maintain multiple signatures. With this feature, you can make dramatic changes to a service program (adding, changing, and removing procedures) without ever touching the programs that use the service program.

PARAMETERS AND SHARED DATA

Prototypes (PR definitions) and procedure interfaces (PI definitions) offer many advantages when you're passing data between modules and programs. For example, they avoid runtime errors by giving the compiler the ability to check the data type and number of parameters. Prototypes also let you code literals and expressions as parameters, declare parameter lists, and pass parameters by value and by read-only reference as well as by reference.

Generally, if the procedure interface for a program or procedure does not appear in the same source member as the calling code, the calling module should include a prototype for that called program or procedure. The prototype need not appear in the same source member as the program or procedure code itself.

Store Prototypes in /COPY Members

Code a /Copy member to contain prototypes for all or most of your reused procedures. Then include a reference to that /Copy member in each module that uses those prototypes. This practice saves you from typing the prototypes each time you need them and reduces errors. An alternative to having one large /Copy member is to group related prototypes in individual /Copy members. Even if you won't use all the prototypes in the /Copy member, remember that there's no runtime penalty for having unused prototypes during the compile.

Include constant declarations for a module in the same /Copy member as the prototypes for that module. If you then reference the /Copy member in any module that refers to the called module, you've effectively globalized the declaration of those constants.

Protect Parameters from Unintended Changes

In a modular programming environment, you might not always know what a called program or procedure will do to a parameter's value. To prevent unintentional changes to a parameter, you should generally pass parameters by value when calling a procedure or by read-only reference (Const) when calling a program:

```
Dcl-pr Celsius Dec(5:0);
  *N Dec(5:0);
End-pr;

Dcl-pr Updcust Extpgm('AR003');
  *N Dec(5:0) Const;
  *N Dec(7:0) Const;
End-pr;
```

Only in those cases where the caller truly needs to see any changes to a parameter, such as in the case of an error message, should you pass a parameter by reference.

Not only do these parameter-passing methods (value or read-only reference) avoid accidental changes to parameter values, but they also allow the caller to pass literals or expressions as parameters, making the ILE RPG code more flexible:

```
Metrictemp = Celsius(Englishtemp);
Metrictemp = Celsius(212);
Metrictemp = Celsius(Englishtemp + 50);
```

Use IMPORT and EXPORT Only for Global Data Items

The Import and Export keywords let you share data among the procedures in a program without explicitly passing the data as parameters. In other words, they provide a *hidden interface* between procedures. Limit use of these keywords to data items that are truly global in the program, usually values that are set once and then never changed. To share data between modules, you should usually pass parameters instead of using Import and Export to share the data.

INDICATORS

Historically, indicators have been an identifying characteristic of the RPG syntax, but they are fast becoming relics of an earlier era. Reducing a program's use of indicators may well be the single most important thing you can do to improve the program's readability.

Eliminate Numbered Indicators

ILE RPG completely eliminates the need for conditioning indicators and resulting indicators, and it does not support them in free-form specifications. The indicator data structure (Indds keyword) and several functions render obsolete the predefined numbered indicators. Use the file I/O functions (e.g., %Eof, %Found, %Error) and the (E) operation code extender to indicate file exception conditions.

If you must use indicators, name them. ILE RPG supports a Boolean data type (N) that serves the same purpose as an indicator. You can use the Indds keyword with a display file specification to associate a data structure with the indicators for a display or printer file; you can then assign meaningful names to the indicators.

Include a description of any indicators you use. Even after you eliminate numbered indicators, a handful of predefined indicators may remain (e.g., L0–L9 level break indicators, or the U1-U8 external indicators). It's especially important to document these indicators because their purpose isn't usually obvious by reading the program. The preface is a good place to list them.

Always Qualify File I/O Functions

The %Found, %Eof, %Equal, and %Status functions allow you to specify a file name with which to associate the functions' return value. To avoid ambiguous file exception reporting, always include the file name with these functions:

```
%Eof(Customers)
```

instead of

```
%Eof
```

The %Error function does not provide for a file name qualifier. Check the %Error function immediately after executing an operation with the (E) extender, or use the %Status function instead.

STRUCTURED PROGRAMMING TECHNIQUES

Give those who follow you a fighting chance to understand how your program works by implementing structured programming techniques at all times. Use the structured operation codes—If, Dou, Dow, For, and Select/When/Other—instead of their older fixed-format ancestors. Do not use conditioning indicators to execute loops or groups of code. And avoid using Goto, Cab*xx*, or Comp. Employ Iter to repeat a loop iteration, and use Leave or Leavesr for premature exits from loops or subroutines, respectively.

Perform Multipath Comparisons with Select/When/Other Groups

Deeply nested If/Else code blocks are hard to read and result in an unwieldy accumulation of Endifs at the end of the group. Elseif improves the situation somewhat, but Select/When/Other is usually a better and more versatile construct. The same advice goes for the obsolete Cas*xx* opcode; use Select/When/Other instead.

CHARACTER STRING PROCESSING

IBM has greatly enhanced RPG's ability to easily manipulate character strings. Many of the tricks necessary with earlier versions of RPG are now obsolete. Modernize your source by exploiting these new features.

Use a Named Constant to Declare a String

Instead of storing a string constant in an array or table, declare it in a named constant. Declaring a string (such as a CL command string) as a named constant lets you refer to it directly, rather than forcing you to refer to the string through its array name and index. Use a named constant to declare any value that you don't expect to change during program execution.

Avoid using arrays and data structures to manipulate character strings and text. Use expressions and functions (e.g., %Subst, %Replace, %Editc) instead.

Use Variable-Length Fields to Simplify String Handling

You can process variable-length fields by using simple expressions. Not only does the code look better (eliminating the %Trim function, for example), but it's also faster than using fixed-length fields. For example, use this code

```
Dcl-s File     Varchar(10);
Dcl-s Library  Varchar(10);
Dcl-s Member   Varchar(10);
Dcl-s Qualname Varchar(33);

Qualname = Library + '/' + File + '(' + Member + ')';
```

instead of the following:

```
Dcl-s File     Char(10);
Dcl-s Library  Char(10);
Dcl-s Member   Char(10);
Dcl-s Qualname Char(33);

Qualname = %Trim(Library) + '/' +
           %Trim(File) + '(' +
           %Trim(Member) + ')';
```

Use variable-length fields as parameters to every string handling subprocedure (passing by value or read-only reference), as well as for work fields where appropriate.

COMMENTS

Good programming style can serve a documentary purpose in helping others understand your source code. But use comments judiciously. If you practice good code construction techniques, you'll find that less is more when it comes to commenting the source. Too many comments are as bad as too few. Here are some specific commenting guidelines.

Use Comments to Clarify, Not Echo, Your Code

Comments that merely repeat the code add to a program's bulk but not to its value. In general, use comments for just three purposes:

- to provide a brief program or procedure summary
- to give a title to a subroutine, procedure, or other section of code
- to explain a technique that isn't readily apparent by reading the source

Always include a brief summary at the beginning of a program or procedure. This prologue should include the following information:

- the program or procedure title
- a brief description of the program's or procedure's purpose
- a chronology of changes that includes the date, programmer name, and purpose of each change
- a summary of indicator usage
- a description of the procedure interface (the return value and parameters)
- an example of how to call the procedure

Use Marker Line Comments to Organize Code

You can employ *marker line* comments to divide the major sections of your program. For example, you should definitely section off with lines of dashes (-) the declarations, the main procedure, each subroutine, and all subprocedures. Identify each section for easy reference:

```
// ------------------------------------------------------ Prototypes
Dcl-pr Dayofweek Dec(1:0);
  *N Date;
End-pr;

// ------------------------------------------- Standalone variables
Dcl-s Daynbr Dec(5:0);

// ---------------------------------------------------------------
// Main processing routine
// ---------------------------------------------------------------

// Calculate total pay for employee
Chain(ne) EmployeeID Employees;
If %Found(Employees);       // If employee active, calculate total pay
  Eval(h) TotalPay = (RegHours * Rate) + (OvtHours * Rate * 1.5);
Endif;
```

Use blank lines to group related source lines, and make them obvious. In general, you should use completely blank lines instead of blank comment lines to group lines of code, unless you're building a block of comments. Use only one blank line, though; multiple consecutive blank lines make your program hard to read.

Avoid Using Positions 1–5

The original RPG syntax, which was oriented to using punched paper cards, used positions 1–5 to sequence program line numbers. In ILE RPG, these columns are commentary only. You can use them to identify changed lines in a program or structured indentation levels, but be aware that these columns may be subject to the same hazards as right-hand comments.

Also leave positions 6 and 7 blank for free-format coding. All code should appear between columns 8–80.

AVOID OBSOLESCENCE

RPG is an old language. After nearly 60 years, many of its original, obsolete features are still available. Don't use them.

Eliminate Obsolete Operation Codes

How do you identify an obsolete operation? Simple answer: if the free-form specification doesn't support it, the operation code is obsolete. In introducing the free-form specification, IBM simplified RPG by paring the number of supported operation codes in half. Appendix A summarizes the preferred substitutions for those operations that you shouldn't use anymore.

Choose Functions over Operation Codes

If a function offers the same capability as an operation, use the function instead of the operation. With some operations, you can substitute a function for the operation and use the function within an expression. Functions are preferable if they offer the same capability as the operation codes.

Avoid Program Described Files

Instead, use externally defined files whenever possible.

Use Native Date Data Types to Process Dates

Eliminate the clever date and time routines that you may have gathered and jealously guarded over the years. The RPG date functions (e.g., %Date, %Diff, %Subdt, %Days) are more efficient, more clear, and more modern. Even if your database includes dates in legacy formats, you can use the date functions to manipulate them.

Avoid Programming Tricks

Such maneuvers aren't so clever to someone who doesn't know the trick. If you think you must add comments to explain how a block of code works, consider rewriting the code to clarify its purpose or hiding the code's complexity in a procedure. Use of the obscure *bit-twiddling* functions (%Bitand, %Bitnot, %Bitor, %Bitxor) may be a sign that your source needs simplifying.

FINAL ADVICE

Sometimes good style and efficient runtime performance don't mix. Wherever you face a conflict between the two, choose good style. Hard-to-read programs are difficult to debug, maintain, and get right. Program correctness must always win out over speed. Keep in mind these admonitions from Brian Kernighan and P.J. Plauger's *The Elements of Programming Style* (McGraw-Hill; 2nd edition, 1978):

- Make it right before you make it fast.
- Make it clear before you make it faster.
- Keep it right when you make it faster.
- Write clearly—don't sacrifice clarity for efficiency.