RPG IV: Subprocedures Beyond the Basics

Techniques to Leverage Subprocedures

OCEAN Technical Conference
Catch the Wave

Jon Paris
jon.paris@partner400.com
www.Partner400.com

Your Partner in AS/400 and iSeries Education

In this presentation we have attempted to gather together some of the information that we have collected while using and teaching RPG IV subprocedures and prototyping. We hope you will find it useful. If you have your own tips or techniques on the subject, or if there are areas not covered here that you would like to know more about, we would love to hear from you. Please e-mail us at the address given in this handout.

The author, Jon Paris, is co-founder of Partner400, a firm specializing in customized education and mentoring services for AS/400 and iSeries developers. Jon's career in IT spans 30+ years including a 10 year period with IBM's Toronto Laboratory. Jon now devotes his time to educating developers on techniques and technologies to extend and modernize their applications and development environments.

Together with his partner, Susan Gantner, Jon authors regular technical articles for the IBM publication, *eServer Magazine, iSeries edition*, and the companion electronic newsletter, *iSeries EXTRA*. You may view articles in current and past issues and/or subscribe to the free newsletter at: eservercomputing.com/iseries.

Feel free to contact the author at: Jon.Paris @ Partner400.com

Unless otherwise noted, all features covered in this presentation apply to all releases of RPG IV. Where a function was added in a later release, I have attempted to identify it. For example V3R2+ indicates that the feature is available at V3R2, V3R6, V3R7, etc.

Another Date Routine - FmtDate

Let's use this new routine to review the basics

```
  * FmtDate    - Formats a "pretty" date for any date
  *            - Uses:   DayName
  *            - Input:  WorkDate (Date field in *Eur format)
  *            - Return: Date string (50 characters)

 P FmtDate          B                   Export
 D                  PI             50
 D   InpDate                         D   Const DatFmt(*Eur)

 D Suffix           S               2

 D Month            S               2  0
 D Day              S               2S 0
 D Year             S               4A

 D MonthData        DS

 D Values                         108    Inz('January  February March    +
 D                                            April    May      June     +
 D                                            July     August   September+
 D                                            October  November December')

 D MonthName                        9    Overlay(MonthData) Dim(12)
```

Before we get into completely new areas, we will use this example of a general purpose date routine to review the basics of subprocedure construction.

This routine is supplied as an additional example of the kind of reusable routines that are easily built using RPG IV subprocedure support. Combined with DayName and DayOfWeek** it provides a starting point for a library of date related functions.

For simplicity I tend to keep all of my related functions together in a single source. i.e. this routine would be in the same source member as the other date routines. Of course the prototypes would be in a separate source member and would be copied (/COPY of course!) into this source and that of any program that wished to use this function.

** If you don't already have the code for these subprocedures you can find them on one of the following notes pages.

FmtDate - C-Specs

Partner400

The code below could be replaced by a table of suffixes

- It would contain an entry for each of the 31 possible day numbers
- This would simplify the logic but be much more boring to read !!

```
  * Extract the day portion of the day and work out it's suffix
 C                   Extrct    InpDate:*D    Day

 C                   Select

 C                   When      ( Day > 3 And Day < 21 ) Or
 C                             ( Day > 23 And Day < 31 )
 C                   Eval      Suffix = 'th'

 C                   When      Day = 1 Or Day = 21 Or Day = 31
 C                   Eval      Suffix = 'st'

 C                   When      Day = 2 Or Day = 22
 C                   Eval      Suffix = 'nd'

 C                   When      Day = 3 Or Day = 23
 C                   Eval      Suffix = 'rd'

 C                   EndSl
```

FmtDate - C-Specs

Partner400

How to get the name of the day ?

- Use the DayName procedure from the Procedures Basics session
  - Procedures using procedures using ......

For releases earlier than V4R4

- You will need to replace %Char in the following code by using a combination of %Trim together with %EditC or %EditW

```
  * Now get the Month and Year portions then format the return value

 C                   Extrct    InpDate:*M    Month

 C                   Extrct    InpDate:*Y    Year

 C                   Return    %TrimR(DayName(InpDate)) + ' the '

 C                             + %Char(Day) + Suffix

 C                             + ' of ' + %TrimR(MonthName(Month))

 C                             + ', ' + Year

 P                 E
```

V5R1 Update

Here is the same code this time using V5R1 options

```
// Extract the day portion of the day and work out it's suffix

  Day = %SubDt(InpDate : *D);

  Select;
    When ( Day > 3 And Day < 21 )  Or  ( Day > 23 And Day < 31 );
      Suffix = 'th';
    When (Day = 1)  Or  (Day = 21)  Or  (Day = 31);
      Suffix = 'st';
    When (Day = 2)  Or  (Day = 22);
      Suffix = 'nd';
    When (Day = 3)  Or  (Day = 23);
      Suffix = 'rd';

  EndSl;

// Now format and return the date string

  Return %TrimR(DayName(InpDate)) + ' the ' + %Char(Day) + Suffix
             + ' of ' + %TrimR( MonthName( %SubDt(InpDate : *M) ) )
             + ', ' + %Char( %SubDt(InpDate : *Y) );
```

This version takes advantage of some of the new V5 features. In particular the %SUBDT BIF and the free form coding. As you can see the whole routine is not much more than a huge RETURN statement.

If we had wanted to, we could have done the whole thing on the RETURN. How? Well that is left as an exercise for the reader! Here's a hint, we would need to code an array for the suffixes.

An additional benefit of this approach is that because of the use of %SUBDT, the two workfields Month and Year are not required and their definitions can be removed.

Those of you looking for the source code for DayOfWeek and DayName will find them on the following pages.

Here is the code for DayOfWeek. We will not be going through it in the session so if you have any questions please feel free to e-mail me. The code for DayName is on the page below.

```
  * DayOfWeek - Calculates day of week (Mon = 1, Tue = 2, etc.) for any date
  *           - Input:  WorkDate (Date field in *USA format)
  *           - Return: Single digit numeric

 P DayOfWeek       B                   Export

  * Procedure interface (PI) definition

 D                 PI             1S 0
 D WorkDate                        D

  *     The base date AnySunday can be set to the date of ANY valid Sunday
  *     (If Sunday is Day 1 for you then adjust the base date etc. accordingly)
 D AnySunday       S               D   INZ(D'04/02/1995')
 D WorkNum         S              7  0
 D WorkDay         S              1S 0

 C     WorkDate      SubDur    AnySunday     WorkNum:*D
 C     WorkNum       Div       7             WorkNum
 C                   MvR                     WorkDay

  * This version changed from the original to demonstrate multiple 'Return's

 C                   If        WorkDay < 1
 C                   Return    WorkDay + 7
 C                   Else
 C                   Return    WorkDay
 C                   EndIf

 P DayOfWeek       E
```

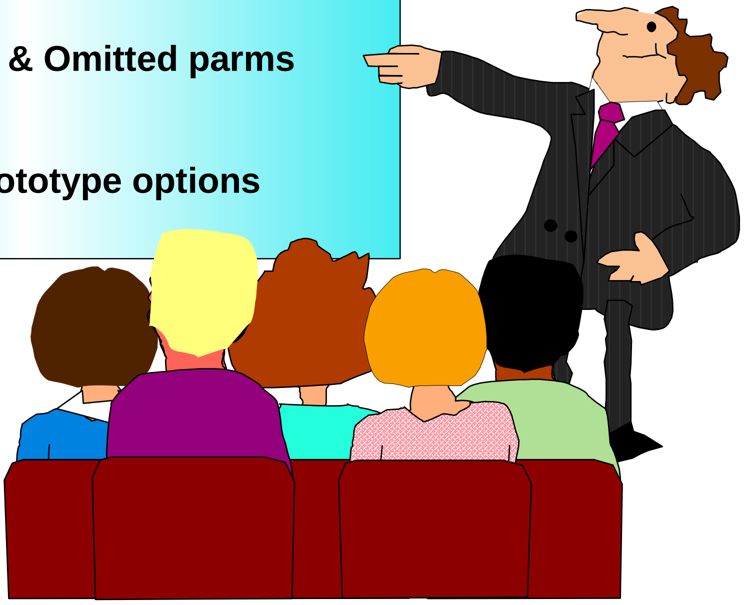
```
  * DayName   - Calculates alpha day name for any date
  *           - Uses:   DayOfWeek
  *           - Input:  WorkDate (Date field in *USA format)
  *           - Return: Alpha day name (9 characters)
  * NOTE: Don't forget to change the array if you change DayOfWeek
  *       to use a day other than Monday as Day 1!


 P DayName         B                   Export

 D                 PI             9
 D WorkDate                        D

 D                 DS
 D DayData                       42    Inz('Mon   Tues  Wednes+
 D                                          Thurs Fri   Satur Sun   ')
 D DayArray                       6    Overlay(DayData) Dim(7)

 C                   Return    %TrimR(DayArray(DayOfWeek(WorkDate)))
 C                             + 'day'

 P DayName         E
```

Parameter Options

Passing by value

Optional & Omitted parms

Other prototype options



In this section we will review some of the parameter options available. Please note that many of these options are not restricted to subprocedures. For example the specification of optional parameters can be done for any type of program, procedure, or function call.

On the other hand, the option to pass parameters by value (that is to say that a copy of the actual data is passed to the called procedure) is only available for procedure and function calls.

There are a number of other parameter options available which we may touch on briefly during the session.

Passing parameters by VALUE

An alternative approach to passing by reference

- A copy of the actual data is passed to the called procedure
  - Not a pointer to the data as when passing by reference
- Like CONST it can accommodates mismatches in data definition
  - For example accepting an *ISO date when a *EUR format is expected
- Since a copy is passed the callee is allowed to change the data
- It only applies to bound calls
  - i.e. It cannot be used when calling a *PGM object

```
  * Prototype for DayOfWeek procedure
D DayOfWeek          PR             1S 0
D                                    D   VALUE DATFMT(*EUR)
     :               :
D StartDate          S               D   DATFMT(*ISO)
     :               :
  * Adjust start date if it falls on a Sunday
C                   If        DayOfWeek(StartDate) = 1
C                   AddDur    1:*D          StartDate
```

The keyword VALUE has a similar effect to CONST. The difference is that when VALUE is specified a copy of the data is passed to the called procedure. This is known as passing parameters by value. The normal method on the AS/400 is known as passing parameters by reference, which means that a pointer to the data is passed.

Passing by value is the method used by C functions, it can also be useful in our own coding if we want to ensure that the called routine cannot affect us by changing the parameter's value. In this regard it is similar to using a PARM with Factor 2 and Result.

Since OS/400 programs (*PGM objects) can only receive parameters passed by reference (i.e. a pointer) they cannot be called in this way. Only subprocedures and procedures called with a CALLB can use the VALUE keyword.

As with CONST, the VALUE keyword allows the compiler to accommodate mismatches in definitions of parameters between the calling and called programs or procedures. When you use this keyword, you are specifying that the compiler is to make a copy of the data prior to sending it. In doing so the compiler will perform any conversions needed before passing the parameter.

Using the VALUE Keyword

```
D EndOfMonth      Pr                D   DatFmt(*USA)
D   USADate                         D   DatFmt(*USA) Value

D Today           S                 D   Inz(*Sys)
D EndDate         S                 D

C                   Eval      EndDate = EndOfMonth(Today)
C     EndDate       Dsply
C                   Eval      *INLR = *On

P EndOfMonth      B
D                 PI                D   DatFmt(*USA)
D   InpDate                         D   DatFmt(*USA) Value

D TempDay         S                2  0

 * Advance the date passed to us by one month
C                   ADDDUR    1:*M          InpDate

 * Subtract day number from the date to reach last day of prior month
C                   EXTRCT    InpDate:*D    TempDay
C                   SUBDUR    TempDay:*D    InpDate
C                   Return    InpDate
P EndOfMonth      E
```

This routine demonstrates the use of the VALUE keyword.

When we use the VALUE keyword, the compiler knows that the parameter passed to it is a copy of the original data. Knowing that it , it will permit changes to be made to it. This saves us from having to define a working copy of the date within the subprocedure and ensures that the original value cannot be changed by the called procedure. Had we used the CONST keyword instead of VALUE, the compiler would have rejected the SUBDUR operation.

Just as with CONST, the VALUE keyword allows the compiler to take care of the fact that an *ISO date was being passed and not the *USA format that is expected. The parameter will be converted from *ISO to *USA while producing the copy.

This procedure also demonstrates that the format of the date returned by the subprocedure does not have to match that of the receiver variable - EndDate in this case which has a format of *ISO.

V5 Version of EndOfMonth

```
D EndOfMonth      Pr              D   DatFmt(*USA)
D   USADate                       D   DatFmt(*USA) Value

D Today           S               D   Inz(*Sys)
D EndDate         S               D

 /FREE
   EndDate = EndOfMonth(Today);
   Dsply EndDate;
   *INLR = *On;
 /END-FREE

P EndOfMonth      B
D                 PI              D   DatFmt(*USA)
D   InpDate                       D   DatFmt(*USA) Value

D TempDay         S              2  0

 /FREE
  InpDate = InpDate + %Months(1);
  Return InpDate - %Days( %SubDt(InpDate : *D) );
 /END-FREE
P EndOfMonth      E
```

Once again, as you can see the V5 version is considerably shorter.

Could you perform the entire calculation on the Return opcode? Yes - but it would mean repeating part of the calculation. i.e. you would have to replace each reference to **InpDate** on the Return with **InpDate + %Months(1)**. The resulting expression might be a little confusing and would probably take longer to run, except possibly at full optimization.

Notes

OPTIONS(*NOPASS)

Indicates that the parameter is optional

- ALL subsequent parameters must also be *NOPASS
- Called procedure must check the number actually passed
- Useful if the procedure is to provide default values

```
D OptionalTest    Pr
D  Parm1                        20A
D  Optional2                    10A   Options(*NoPass)
:                                :
C                    CallP     OptionalTest(FirstParm)
```

```
D ProcInterface   PI
D   Parm1                       20A
D   Optional2                   10A   Options(*NoPass)

D Parm2           S             10A   Inz('DefaultVal')

 * Check for optional parameter and use value if present
C                    If        %Parms = 2
C                    Eval      Parm2 = Optional2
C                    EndIf
```

Note that *NOPASS can only be used when all remaining parameters are also optional and specified as *NOPASS. This is similar to the CL command interface where many parameter values are optional.

Be careful not to confuse *NOPASS with *OMIT - see the following chart

*NOPASS allows a parameter to be omitted completely. The programmer must ensure that the parameter in question is not referenced if it has not been passed. %Parms can be used to establish if the parameter was passed or not.

As shown in the example above, If the parameter is not passed, the called procedure can handle this by creating a local variable which is initialized to the default value. If the parameter is passed it is moved to this variable which is subsequently used in all further processing.

Note that you cannot move a default value into the parameter if it wasn't passed - there's nowhere to put it!! You will get a pointer exception just as you would if you tried to reference the parameter directly.

You should also be aware that attempting to access a parameter (or checking to see if it has a valid address) is not a good idea. You may get a "false positive" due to a pointer being in the right place on the stack. Of course it is not the pointer you want, and you could be corrupting just about anything if you move data to the parm!!

<u>Always, always, always</u>, use %Parms to check the number of parms passed if optional parameters are in use.

Using Options(*NoPass)

- This is a variation that uses an optional parameter
  - If a "Months" parameter is NOT supplied, the routine assumes that the end date of the month in which the date falls is to be returned.
  - If a parameter is supplied, then the routine will return the last day of the month that many months from the date that was passed.
- The input variables can be modifed since they were passed by Value

```
P EndOfMonth       B
D                  PI              D   DatFmt(*USA)
D   InpDate                        D   DatFmt(*USA) Value
D   Months                        3P 0 Options(*NoPass) Value

 * If the optional "Months" value is passed then advance by that
 *   value plus one month otherwise advance the date by one month
C                    If        %Parms > 1
C                    Eval      Months = Months + 1
C                    AddDur    Months:*M     InpDate
C                    Else
C                    AddDur    1:*M          InpDate
C                    EndIf
 :                   :         :             :
```

By modifying our example to use an optional parameter we have enhanced its functionality. It can now advance not just to the end of the month but to the end of the month "n" months from now.

Perhaps as important is the fact that we have made the change without impacting its original behaviour! The routine tests to see if the second parameter is present, and if it is not, it uses the default value of 1. Since this is the behaviour of the original routine no change would be required in any program that was using the it.

You might like to consider what changes could be made to the subprocedure to allow the first parameter (i.e. the date) to be optional. If no date were passed we could use the current date as the default. The implementation of thi is left as an exercise for the student. e-mail me (Jon.Paris@ Partner400.com) if you get stuck for an answer.

OPTIONS(*OMIT)

Allows the special value *OMIT to be used as the parm

- It cannot really be omitted
- The called procedure must check for this special value
- It can do so by checking to see if the pointer is null
  - This cannot be done if the parameter was passed by "Constant Reference"
    - I.e. the keyword CONST was coded on the prototype
  - In this case the API CEETSTA must be used
    - Unless you are using V5R1 or later where this restriction is removed

```
D ParmTest        Pr
D  NormalParm                      10A
D  OmitParm                        10A   Options(*Omit)
D  NormalParm                      10A
:
C                   CallP     ParmTest(Parm1 : Parm2 : Parm3)
:
C                   CallP     ParmTest(Parm1 : *Omit : Parm3)

 * Test for omitted parms in the called procedure like this
C                   If        %Addr(Parm2) = *Null
```

The option *OMIT can be used for parameters which are not mandatory but which occur in the middle of a parameter sequence and therefore cannot be designated as *NOPASS.

*OMIT is only allowed for parameters that are passed by reference, including those with the CONST keyword. More on this in a moment.

Several system APIs use this option - the parameter is designated as "can be omitted"

When the option is specified you can either pass an appropriate parameter (in our example a 10 character field) or the special value *OMIT

Note that the parameter will still count in the number of parameters passed and the called program will need to test to see if the parameter was actually passed. Any attempt to reference the parameter when *OMIT was passed will result in an error. There are two ways to do this:

- Compare the %Addr of the parameter to *Null. For releases prior to V5R1 this method cannot be used if the parameter was designated as CONST.
- You can use the API CEETSTA (The RPG Programmers Guide provides a brief example.) to determine if a parameter was actually passed.

Other OPTIONS

OPTIONS(*VARSIZE)

- The parameter can be shorter than specified in the prototype
  - Or in the case of an array can have fewer elements
- It must be passed by reference (i.e. no VALUE keyword)
- Applies to character fields and arrays only

OPTIONS(*STRING)

- Mainly used when calling C functions
- Can only be used to describe data pointers
  - Allows either a pointer or a string expression as the parameter
  - If an expression is passed, the string is copied to a temporary area, null terminated and a pointer to that area passed

```
D VarSizeTest     Pr
D  VarSize1                    20A   Options(*VarSize)
:                               :
D TenLong         S            10A
:                               :
C                   CallP     VarSizeTest(TenLong)
```

The Options(*VarSize) keyword can be useful in subprocedures, but is most useful when prototyping program calls. In the case of arrays "shorter" means that the array has fewer elements than specified in the prototype.

Starting with V4R2, RPG IV supports the use of varying length character fields and these are a much better option when building subprocedures that must accept character fields of different lengths. We hope to demonstrate this to you on the next chart.

Exploiting Varying Length fields

The procedure prototype and examples of its use

- The prototype should of course be /COPY'd in !!

```
D CenterFld       Pr            256A   Varying
D  Fld                          256A   Varying Const

D Field20         S              20
D Field40         S              40
D Field50         S              50
D CenterCon       C                     'Center this data'

C                   MoveL     CenterCon     Field40
C                   Eval      Field40 = CenterFld(Field40)
C     Field40       Dsply
C                   MoveL     CenterCon     Field20
C                   Eval      Field20 = CenterFld(Field20)
C     Field20       Dsply
C                   MoveL     CenterCon     Field50
C                   Eval      Field50 = CenterFld(Field50)
C     Field50       Dsply

C                   Eval      *InLR = *On
```

You may have noticed in the prototype that both the return value and the parameter for this subprocedure are specified as varying length fields (keyword VARYING). Yet none of the fields passed as parameters to the routine are varying length. So how does this work?

- For the parameter, the CONST keyword takes care of things. When a fixed length field is passed, the compiler will generate a temporary varying length field, copy the field to this temporary, and pass the temporary to the subprocedure. The compiler will store the original length of the field as the current length of the varaible length field.
- The compiler will handle the conversion of the returned value, copying the data into the target field as if it were a fixed length field of the same length as its current length.

This is a case where we don't actually want to use a parameter of the type defined. If we were to pass a variable length field, the routine would probably not produce the desired result. It is designed to work with fixed-length fields.

Varying Length fields ... (Cont.)

This is the subprocedure itself

```
P CenterFld       B
D                 PI           256A   Varying
D  Fld                         256A   Varying Const
D Len             S              5I 0
D Pad             S            256A   Inz Varying
D Temp            S            256A   Inz Varying

  * Capture length of original input field
C                   Eval      Len = %Len(Fld)
  * Strip trailing blanks
C                   Eval      Temp = %TrimR(Fld)
  * Set Pad to required length - content is spaces due to INZ keyword
C                   Eval      %Len(Pad) = %Int((Len - %Len(Temp)) / 2)
  * Now just return the two "glued" together
C                   Return    Pad + Temp
P                 E
```

Let's look at each operation in the subprocedure to see how it works.

The first Eval captures the length of the original input parameter. This works because when the compiler copies the parameter into the temporary variable length field the length of that field will be set to the length of the original field (i.e. no trimming takes place on the original content). We will use this information later in determining how many spaces need to be inserted into the front of the field to center the content.

The next task is to strip any trailing blanks from the input. If you want the routine to ignore leading spaces also you should change the %TrimR to a %Trim. The result is placed in the field Temp.

We now calculate the number of spaces that we need to insert to center the field. This is done by subtracting the length of the field Temp (which contains the input field stripped of trailing spaces) from the length of the original parameter field. This is then divided by 2 to calculate the number of spaces that must be inserted. Note that if you want any "odd" spaces to be on the left you should change the calculation to (%Int((Len - %Len(Temp)) + 1) / 2). The result of the calculation is used to set the length of the field Pad, which will contain spaces.

All that remains is to return the original field prefixed by the pad field.

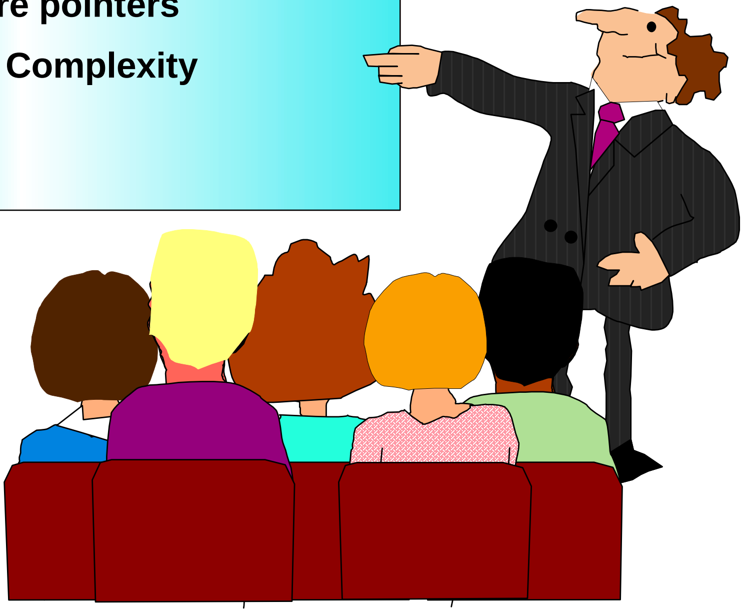
What else is there?

Partner400

Dynamic and Static Storage

Procedure pointers

Masking Complexity



In this section we will briefly discuss a number of topics that relate to Subprocedures.

Procedure pointers.
These can provide a level of dynamic call support for bound calls i.e. they permit us to call using a variable name. The variable is a procedure pointer into which we can place the address of any procedure that we wish to call. Note that it is the address and not the name of the procedure we will be using. They also facilitate "call back" processing which can be a useful means of allowing a generalized function access to a program specific error routine.

Dynamic and Static Storage
Subprocedures support both dynamic and Static storage. Conventional RPG program support only Static - we will look at the differences.

Masking Complexity
We will not be going into this in detail, just introducing you to the idea. The RPG Redbook mentioned at the end of the handout contains more information if the topic interests you. The basic notion is that subprocedures can be used to mask the complexity of certain functions and to simplify the main line logic. We want the main line to read like pseudo code. Rather than have a comment such as "Retrieve Customer data" followed by a number of READ, SETLL, CHAIN or whatever operations, we'd much rather see **CustData = GetCustomerInfo(CustNum)**. Much simpler to follow and far less susceptible to errors being introduced during maintenance. Similarly the complexity of subfile handling can be "hidden" from a junior programmer by "wrapping" it in a subprocedure so that in the mainline code what appears is a reference to a procedure such as: ClearSubfile or DisplaySubfile.

Another area where such techniques can be used is with APIs, so that not everyone in the shop has to understand them. They only need to understand the simplified interface provided by the subprocedure.

Static vs. Automatic Storage

Local data fields use automatic storage by default

- Can be overridden with the STATIC keyword on the D spec
  - In other words, local data can be either static or automatic

What's automatic storage?

- Storage that exists only for the duration of the procedure call
- Goes away (is de-allocated) when procedure returns to its caller

Static storage is the only type for global fields in RPG

- LR indicator controls when global fields are reinitialized
  - Local fields -- even if declared STATIC -- are *not* reinitialized after LR

Why use automatic storage?

- Automatic "clean up" of fields without the overhead of LR
- More efficient - storage not used unless / until the procedure is called
- It allows for recursive calls

Data stored in automatic storage goes away when the procedure returns to its caller. Upon subsequent calls to the procedure, automatic fields "lose" their values. Data stored in static storage remains between calls to the procedure until the Activation Group where the procedure is activated is reclaimed.

Recursion (the ability for a subprocedure to be called multiple times in the same invocation stack -- in fact, for a subprocedure to call itself!) is made possible because of automatic storage. Each call to the subprocedure gets a "fresh" set of local automatic storage fields. This is why subprocedures may be called recursively, but main RPG procedures cannot.

Automatic storage can be considerably more efficient than static storage since it is only allocated if and when the subprocedure is called. Think about static binding and the fact that many procedures are often activated at once in a job! Remember that all the procedures in all the Service Programs referenced (directly or indirectly) by an ILE program are allocated immediately on the first call in the job to that ILE program. If that represents a lot of procedures (either main procedures or subprocedures) all static fields will be allocated and initialized by the system right away. In many cases, many of those procedures may never be called in this particular job. However, their static storage must always be allocated and remain allocated until the program's Activation Group is reclaimed. Use of automatic storage reduces this potential "overuse" of memory by allocating only what is needed and only for the duration that it is needed. In "memory-heavy" applications, this could have a noticeable impact on application performance and system efficiency.

Example: Automatic & Static

What are the values displayed each time?

```
H DftActGrp(*NO)  ActGrp('QILE')

D ProcAuto        PR

C                   Do        5

C                   CallP     ProcAuto

C                   EndDo
C                   Eval      *INLR = *On

P ProcAuto        B

D                 PI

D CountStat       S              3  0 Static  Inz
D CountAuto       S              3  0 Inz

C                   Eval      CountStat = CountStat + 1
C                   Eval      CountAuto = CountAuto + 1
C      'CountStat='  Dsply                  CountStat
C      'CountAuto='  Dsply                  CountAuto

P                 E
```

To test if you really understand automatic and static storage, can you predict what values will be displayed when the main procedure in this sample code is called?

The value of CountStat increases by 1 each time it is displayed.

The value of CountAuto never increases. It is displayed as 1 every time!

Now, for the tougher question:

Assume this main procedure was immediately called a second time in the same job. What value would be displayed for CountStat on the first call to the ProcAuto subprocedure ??

Procedure Pointers

Specified by adding keyword PROCPTR to the definition

They allow you to call a variable target

- Procedure pointer must be set before the call
  - Either by using %PAddr to supply initalization values as in this example
  - Or by using APIs
    - APIs can only reference procedures in Service Programs
    - %PADDR initialization can be to any procedure
    - So you can make a "Service Program" out of any program object

All procedures must have a compatible interface

```
D Compute         Pr             15P 5 ExtProc(ProcToUse)
D  Factor1                       15P 5 Value
D  Factor2                       15P 5 Value

D  ProcToAdd                       *   ProcPtr Inz(%PAddr('ADD'))
D  ProcToSub                       *   ProcPtr Inz(%PAddr('SUB'))
D  ProcToMul                       *   ProcPtr Inz(%PAddr('MUL'))
```

This is one of the rare situations where the prototype used in the call is not the same as that used for the individual subprocedures. They do however need to have the same format.

The prototypes for the actual subprocedures themselves used in this example are as follows:

```
D ADD             Pr             15P 5
D  Factor1                       15P 5 Value
D  Factor2                       15P 5 Value

D SUB             Pr             15P 5
D  Factor1                       15P 5 Value
D  Factor2                       15P 5 Value

D MUL             Pr             15P 5
D  Factor1                       15P 5 Value
D  Factor2                       15P 5 Value
```

The actual subprocedure that gets called will depend on the contents of the procedure pointer ProcToCall which will be set during the program logic. Notice that in the Compute prototype the name of the procedure pointer field is not contained in quotes. This is what identifies it as a variable and not the name of a subprocedure.

Procedure Pointers (Contd.)

```
D Value1                          15P 5
D Value2                          15P 5
D Function                         1A
D Result                          15P 5

  * Decide which proc to call
C                   Select
C                   When      Function = '+'
C                   Eval      ProcToUse = ProcToAdd
C                   When      Function = '-'
C                   Eval      ProcToUse = ProcToSub
C                   When      Function = '*'
C                   Eval      ProcToUse = ProcToMul
C                   Other
C                   Eval      ProcToUse = *Null
C                   EndSl

C                   If        ProcToUse <> *Null
C                   Eval      Result = Compute(Value1 : Value2)
C                   Else
C                   Eval      Result = 0
C                   EndIf
```

Using this method, it is possible to call any procedure in any program or service program.

In the example here, the procedures are presumed to included either in the same compilation unit as the code that invokes them, or bound to it. As an alternative, we can also call procedures in a completely separate program, as long as it provides us with the procedure pointers.

For instance the program could consist of a small main line, followed by the procedures. The mainline contains a series of initialized pointers - each identifying a particular procedure. The process is this:

- When the mainline is called, it passes the pointers to its caller
  - It could make decisions about which pointers to supply based on the user Id or similar information
- The caller then uses those pointers to invoke the specific function(s) it requires.

Because the target of the initial call is a program (which of course can called by name) this setup allows you to call virtually any procedure in any program.

Procedure pointers are also useful in implementing "callback" programming techniques. The C functions qsort and bsearch are good examples of this. We will look at a brief example later.

Procedure Pointers (Contd.)

This is the source for the ADD subprocedure

- and its associated prototype

The source for the SUB and MUL functions is similar (very!)

Note that the result is computed and returned in one step

```
D ADD             Pr            15P 5
D  Factor1                      15P 5 Value
D  Factor2                      15P 5 Value
:    :                             :
:    :                             :
P ADD             B

D                 PI            15P 5
D  Factor1                      15P 5 Value
D  Factor2                      15P 5 Value

C                   Return    Factor1 + Factor2

P ADD             E
```

The source for the SUB and MUL subprocedures is almost identical to the ADD.

Needless to say one would not bother to do all this work for such trivial tasks as these. The idea here is to show you the principals involved.

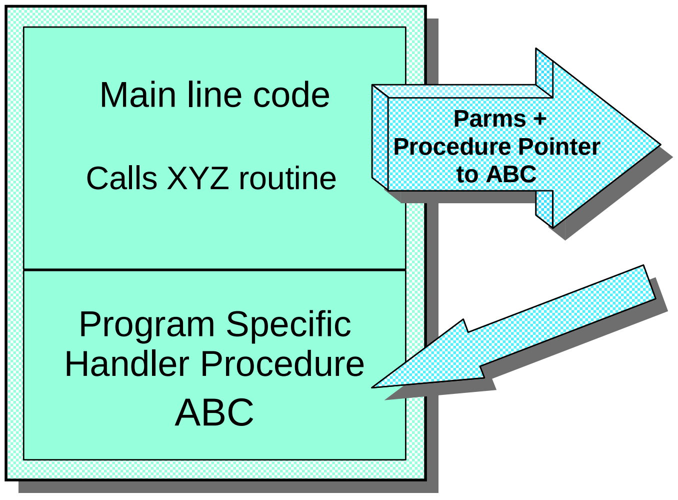
In our example we have hard-coded the initialization values of the procedure pointers. However, because the content of the pointers can be determined programmatically, there are many things that can be done that are far more difficult if not impossible through more conventional means.

- New routines can be introduced
- The same call can target different processing for different users
- Some users can be "locked out" or allowed to use certain functions at specific times.
- etc. etc.

Call Back Processing

User specified code is called from a utility routine

- Useful technique in a number of areas
  - For example to allow a common validation routine to provide program specific error handling


Main line code
Calls XYZ routine
Parms + Procedure Pointer to ABC
Utility Procedure XYZ
Calls "home" using supplied Procedure Pointer
Program Specific Handler Procedure ABC

Once you begin to develop subprocedures, you will inevitably come to a point where it seems that a common routine would be useful, but certain actions related to the processing are very specific to the programs that would use it. Call Back processing is an excellent way of handling this.

In other languages this technique is fairly common. There are a number of examples in the C library. For instance the qsort function uses a user supplied call back procedure to determine which of two elements is the larger. The qsort function itself handles the mechanics of the sort, but requires program specific user code to make the sequencing decisions.

In our example we are using the idea of a generic "is it numeric" validation routine. This is called from our main line program, but any errors that occur will need to be reported by the program. Of course there are many other ways of handling this situation, but this has the advantage that the main logic flow remains relatively "clean". Besides, we are demonstrating he technique here not proposing it as the perfect solution for all programming problems.

Call Back Processing (Cont.)

Partner400

The procedure ValidNum is "called" by the IF opcode

- It is passed the "ordinary" parameters
  - In this case the data to be validated
- And a Procedure Pointer to the required error handling procedure
  - ValidNum will determine if the error routine needs to be called or not

```
D MyErrorCode     Pr
D   InpString                     15A   Varying Value

D ValidNum        Pr               N
D   InpString                     15A   Const Varying
D   ErrorHandler                    *   ProcPtr Value

D TestData        S               15A   Inz('12345.96ABCD')

  * Process valid numbers only - errors handled by MyErrorCode
C                   If        ValidNum(TestData: %PAddr(MyErrorCode))
  ............................................
C                   EndIf
```

Call Back Processing (Cont.)

Partner400

This is the outline of the generic validation routine

- In the event that it detects an error it calls the user supplied procedure

See the notes page for a variation using an optional parm

```
P ValidNum        B
D                 PI               N
D   InpString                     15A   Const Varying
D   ErrorHandler                    *   ProcPtr Value

D UserErrorCode   Pr                    ExtProc(ErrorHandler)
D   OriginalParm                  15A   Varying Value

  * Common routine performs validation etc. here

  * If an error is detected, call the user routine to report it
C                   CallP     UserErrorCode(InpString)
C                   Return    *On

P ValidNum        E
```

Call Back Processing (Cont.)

This is the user specified error handler

- In this case it simply receives a copy of the original parameter data and formats an error message

```
P MyErrorCode     B
D                 PI
D   TestString                   15A   Varying Value
D Message         S              52A
 * Your program specific error routine is coded in this subprocedure
 * If an error is detected, the user supplied routine is called
C                   Eval      Message = 'Field (' + TestString
C                                       + ') is not a valid number'
C     Message       Dsply
C                   Return
P MyErrorCode     E
```

Sometimes it is useful to be able to make the call back procedure optional. For example the routine we are calling may contain default error handling logic and in some cases that will be good enough. In these cases we can make the call back apremeter optional i.e. Options(*NoPass) as shown in this example.

```
P ValidNum        B
D                 PI              N
D   InpString                   15A   Const Varying
D   ErrorHandler                  *   ProcPtr Value Options(*NoPass)

D UserErrorCode   Pr                  ExtProc(ErrorHandler)
D   OriginalParm                15A   Varying Value

D ErrorMessage    C                   'The Default Handler was used'

 * Common routine performs validation etc. here

 * If an error is found, the user error routine is called if supplied
C                   If        %Parms > 1
C                   CallP     UserErrorCode(InpString)
C                   Else
 * No user routine supplied so perform default error handling
C     ErrorMessage  Dsply
C                   EndIf

C                   Return    *On
P ValidNum        E
```

Wrapping and Masking

Don't just think of subprocedures for reusable code

- Some functions are hard to implement/understand

You can "Wrap" APIs (including the C functions) to provide:

- A simplified interface
  - With intelligent defaults
    - And the defaults can be tailored to your shop's requirements
  - Only one person has to understand how the function works
    - The rest just use the new interface
- Re-sequenced parameters
  - The ones most likely to be needed can be placed first
- Generalized error handling
  - With the ability for the calling program to also handle the error

You can also Mask the complexity of your programs

- Using subprocedures to contain the actual logic
  - Leaving the main line to read more like pseudo code

The RPG Redbook "Who Knew You Could Do That with RPG IV?" describes two procedures that "wrap" User Space APIs to achieve the objectives outlined here. You can find more information on the Redbook at the end of this handout.

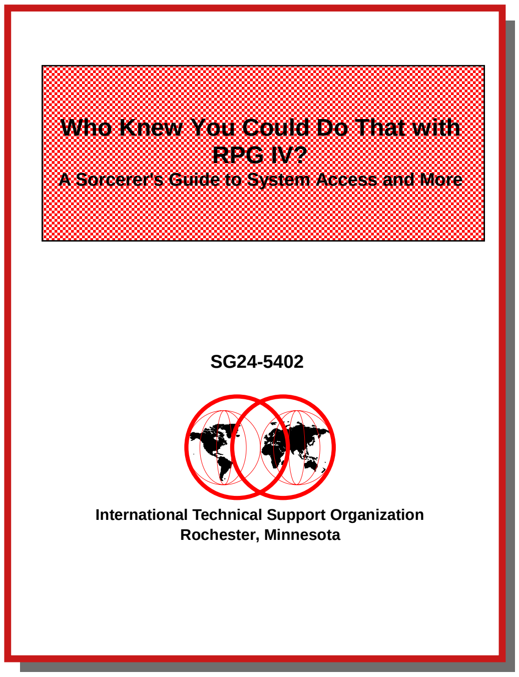
The Redbook example also shows how the procedures are used.

When implementing these kinds of procedures, or indeed any general purpose procedures, it is important that they be documented. My preference is for them to follow the format used by IBM for the RPG manuals, and in particular the pieces describing the Built-in Functions. One of the (few?) advantages to IBM moving to HTML based documentation is that it is a fairly simple matter to add your own documentation to your companies intranet so that is available in the same form as the regular manuals.

For more information:

Check out the RPG Redbook

- Available now - SG24-5402
- Go to www.redbooks.ibm.com
  - You can read it on-line
  - Download the PDF file
  - Or order hardcopy
- Includes worked examples of
  - RPG IV Subprocedures
    - Some of the examples may look familiar!
  - Procedure "wrappers"
  - TCP/IP Sockets
  - CGI programming
  - Using the C function library
  - ILE Error handling
  - and much more ....



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

SG24-5402

International Technical Support Organization
Rochester, Minnesota

Some of the routines in this handout were used as part of the base material which was used in preparing the new RPG Redbook. The book also contains a brief tutorial on ILE as well as pieces on prototyping, and much, much more.

If you read the Redbook and like it, don't forget to use the feedback form at the Redbook web site to let IBM know that you'd like to see more RPG oriented Redbooks!!!