RPG IV Subprocedures

The Basics

OCEAN Technical Conference
Catch the Wave

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

Partner400
Your Partner in AS/400 and iSeries Education

This presentation may contain small code examples that are furnished as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. We therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All code examples contained herein are provided to you "as is". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

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.

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.

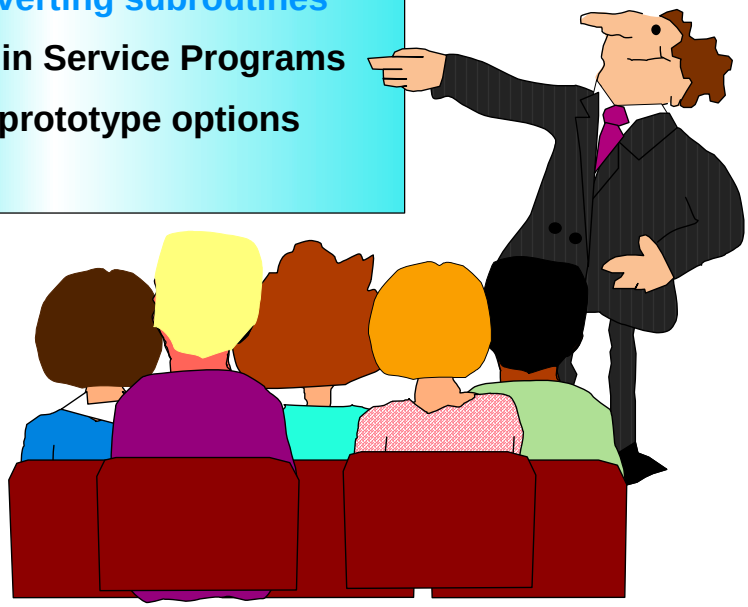
Together with his partner, Susan Gantner, Jon authors regular technical articles for the IBM publication, *eServer Magazine, iSeries edition*, and the companion electronic newsletter, *iSeries EXTRA*. You may view articles in current and past issues and/or subscribe to the free newsletter at: eservercomputing.com/iseries.

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

User Defined Procedures

Partner400

- The basics
  - Converting subroutines
- Procs in Service Programs
- Basic prototype options



In this section we will introduce you to the basics of writing subprocedures in RPG IV

We will start by showing how a simple subroutine can be converted to a subprocedure.

Next we will go on to discuss how subprocedures can be used in Service Programs.

Finally we will look at some of the additional prototyping options available for use with subprocedures.

As we hope to convince you in this section, Subprocedures are probably the biggest single change ever in the history of RPG. They open up whole new ways of building your applications.

Notes

Revision - Built-in Functions (BIFs)

RPG IV supplies a number of BIFs

- Let's use %TRIM as an example
- You pass it a variable (or expression) as a parameter
- It processes the parameter and returns a new string
  - It contains the original string stripped of leading and trailing spaces
  - See the code below for an example of how it works
- Note the terms parameter and returns
  - We will be referring to them later on

Subprocedures work in exactly the same way as BIFs!

```
D TestData        S             20    Inz('      ABCDEFGHI     ')
D Result          S             22

C                   Eval      Result = '(' + TestData + ')'
 * Result now contains '(      ABCDEFGHI     )'

C                   Eval      Result = '(' + %Trim(TestData) + ')'
 * Result now contains '(ABCDEFGHI)           '
```

We're starting off with a very quick review of BIFs since Subprocedures work in a similar manner. The important thing to note is that there is nothing magical about these things. BIFs are designed to take zero or more parameters and to return a result.

For example the BIF %EOF takes a file name as a parameter** and will return an indicator with the value *On or *Off. Because it returns an indicator, we can use it in expressions just as if it were an actual indicator. So **If %EOF(TransFile) = *On** is every bit as valid as **If *In99 = *On**

Other BIFs, such as %TRIM, which we use in the example above, return character strings. Still others return integers (e.g. %SIZE or %ELEM).

IBM adds new BIFs with each release - but they will never supply everything we might ever need, nor will they always work exactly the way we'd like them to. That is one area where subprocedures play a role. We get to write our own functions to fill in the gaps!

** We do know that the filename can be omitted from %EOF but it is bad practice to do so, and it isn't relevant to the discussion here.

Notes

Subprocedures

Partner400

This is the RPG IV name for a Function or Procedure

- If it returns a value it is a Function
  - And it operates in much the same way as an IBM Built-In Function (BIF)
- If it does not return a value it is a Procedure
  - It is simply called with a CALLP and "does stuff" for you

Subprocedures can:

- Define their own Local variables
  - This provides for "safer" development since only the code associated with the variable can change its content
    - More on this later
- Access Files Defined in the Global section
  - By that we mean the main body of the source
- Access Global variables
- Be called recursively

RPG source members can contain multiple subprocedures

Support for subprocedures was added to the RPG IV language in releases V3R2 and V3R6. User written subprocedures in RPG IV allow for recursion (i.e., the ability for the same procedure to be called more than once within a job stream). They also allow for true local variable support (i.e., the ability to define fields within a subprocedure that are only seen by and affected by logic within the bounds of the subprocedure.)

RPG IV subprocedures use prototypes, which are a way of defining the interface to a called program or procedure. In this session, we will concentrate on writing and using RPG IV subprocedures, but you will find that many of the same prototype-writing skills can be applied to access system APIs and C functions.

Note that although a CALLP is used to invoke subprocedures that do not return a value, you should not be mislead into thinking that CALLP means CALL Procedure. It does not - it actually stands for CALL with Prototype.

Most of the time your subprocedures will return a value, but sometimes you'll just want to have it "do stuff". For example a subprocedure named WriteErrorLog might be used to record errors detected by the program. There's not a lot of point in having it return a value to say it did it - after all what are you going to do if it couldn't? Call it again to write another error message? <grin>

When we talk about the "Source Member" we really mean all of the RPG IV source that is processed by the RPG compiler in any one compilation. This would include any source members that were /COPY'd into the main source. You may wonder why we used the term "Source Member" rather than "Program". Traditionally we have tended to equate a source member to a program since the normal RPG/400 approach meant that one source was compiled and this resulted in a program (*PGM) object. With RPG IV each source is compiled into a module (*MODULE), and a number of modules may be combined into a single program.

Major Features of Subprocedures

Subprocedures can use other Subprocedures

- But cannot contain their own "private" subprocedures

When they return a value they are used as functions

- Just as if they were built into the language like IBM's BIFs
- They have a data type and size
  - You can use them anywhere in the freeform calcs where a variable of the same type can be used
    - e.g. In an IF, EVAL, DOW, etc. etc.

A Subprocedure DayOfWeek could be used as shown below

- We will be developing this routine later

```
C           If        DayOfWeek(ADateFld) > 5

C           Eval      WeekDay = DayOfWeek(Today)
```

Subprocedures which return a value are used very much like RPG IV built-in functions, as shown in the examples on this chart. In the example, "DayOfWeek" is the name of an RPG IV subprocedure. In fact we will be building this exact procedure shortly.

It requires a single date field as the input parameter (which in the first example is passed via the field named "ADateFld" and in the second example via the field "Today").

It returns a value, which is a number representing the day of the week (1 through 7). The returned value effectively replaces the function call in the statement. In the first example, that value will be compared with the literal 5. In the second example, the returned value will be placed in the field "WeekDay".

Notes

DayOfWeek Subroutine

```
D InputDate       S               6  0
D DayNumber       S               1  0
 * Variables used by DayOfWeek subroutine
D AnySunday       C                    D'04/02/1995'
D WorkNum         S               7  0
D WorkDay         S               1S 0
D WorkDate        S                 D
       :            :          :
C     *MDY          Move       InputDate     WorkDate
C                   ExSr       DayOfWeek
C                   Move       WorkDay       DayNumber
       :            :          :
 ***** Calculate day of week (Monday = 1, Tuesday = 2, etc.)
C     DayOfWeek     BegSr

C     WorkDate      SubDur     AnySunday     WorkNum:*D
C     WorkNum       Div        7             WorkNum
C                   MvR                      WorkDay
C                   If         WorkDay < 1
C                   Add        7             WorkDay
C                   EndIf
C                   EndSr
```

In this presentation, we will take the subroutine in this program and turn it into a subprocedure. Existing subroutines in programs often make good candidates for subprocedures.

Here we see the traditional use of a standard subroutine, along with all its inherent problems. WorkDate and WorkDay are "standard" field names within the subroutine that we have to use. In effect they are acting as parameters.

We must move fields to/from these "standard" fields to in order to use the subroutine. Once we have turned this into a subprocedure, these additional steps will not be necessary.

The use of common subroutines also forces us to use naming standards to ensure that work fields within the subroutine do not get misused. This can certainly hinder attempts at producing meaningful field names - particularly with RPG III's six character limit!

Notes

Basic Subprocedure

```
D DayOfWeek       PR             1S 0                    5
D  ADate                          D
:
C                   Eval      DayNumber = DayofWeek(InputDate)   1
:                     :        :
P DayOfWeek       B

D DayOfWeek       PI             1S 0                    2
D  InpDate                        D

D AnySunday       C                      D'04/02/1995'
D WorkNum         S              7  0                          3
D WorkDay         S              1S 0

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

C                   If         WorkDay < 1
C                   Add        7              WorkDay
C                   EndIf

C                   Return     WorkDay                 4

P DayOfWeek       E
```

This is the code for our completed subprocedure. Note that we have used the common practice of using /COPY to bring in the prototype code. The /COPY DOWPROTO line will copy a source member which contains the prototype code. We will look at it in detail on a later chart.

Notice the basic sequence of the specifications. The prototype(s) appear at the beginning of the source, along with any other D specs. The P specification that begins the subprocedure appears after the regular C specs. We will look in more detail at the sequence of specifications in this "new style" of RPG program later,

In this example, the three fields with an "S" for Stand-alone fields are local variables available only to the logic in this particular subprocedure. More on what we mean by "local" later.

Notes

Invoking the Subprocedure

Partner400

Converting to a subprocedure allows us to use

DayOfWeek as if it were a built-in function of RPG

- It just doesn't have a % sign in front of the name!

The date to be processed (WorkDate) is passed as a parm

- No need to 'fake' parameters as in the original
  - More on parameter definition in a moment

```
C         Move      InputDate     WorkDate
C         ExSr      DayOfWeek
C         Move      WorkDay       DayNumber
```

```
:           :           :
C         Eval      DayNumber = DayofWeek(InputDate)
:           :           :
```

We call the subprocedure in much the same way as we use a built-in function in RPG. Since it returns a value, we can call it in an expression. The returned day number will be placed in the field called DayNumber. If our subprocedure did not return a value we would invoke it with a CALLP.

Notes

P-specs & the Procedure Interface

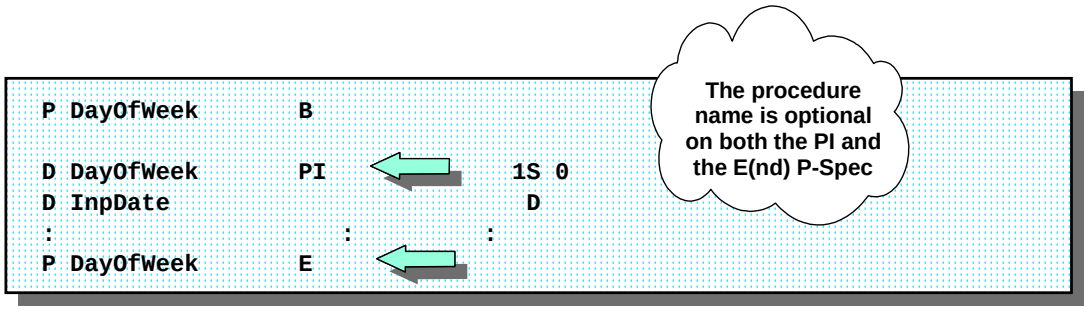
Partner400

Procedures are bounded by P-specs

- A type B(egin) names the procedure
- A type E(nd) is required to complete it

A Procedure Interface (PI) is also required

- The PI line defines the Data type and Size of the procedure
  - The procedure can be used anywhere that a field of the same type and size can be used
- Subsequent lines define any parameters
  - i.e. The PI acts as the procedure's *ENTRY PLIST



Subprocedures begin and end with P specs. In this example, we see the beginning P spec. In a later chart, we will see the ending P spec as well. The beginning P spec contains a B in the position that would contain, for example, a DS on a D spec. The P spec has a very similar layout to the D spec.

The next thing we need is a Procedure Interface, or PI. The procedure interface defines the interface to the procedure -- most significantly, the parameters passed to and/or from the subprocedure. The PI is defined on the D specs, typically as the first D specs in the subprocedure. The PI replaces the need for a *ENTRY PLIST.

The data item defined on the same line as the PI is the return value. (Note: It is possible to have a subprocedure that returns NO value. A subprocedure can return, at most, one value.)

The data item(s) that follow the PI with nothing specified in the column where the PI (in this example, the WorkDate field) is are the input parameters into the subprocedure.

The data items following WorkDate are all local fields to the subprocedure. You can tell the end of the procedure interface and by the appearance of something (in this example, an "S") in the column where PI was specified.

Notes

DayOfWeek Subprocedure

Partner400

Subprocedures allow us to define local variables

- They can ONLY be referenced within the subprocedure

Much safer than the traditional RPG approach

- The writer of the procedure can protect all of the variables
  - Only those that should be changed can be changed !!!!

Note that the constant 'AnySunday' terminates the PI

- Just as it would terminate a Data Structure

```
D DayOfWeek     PI              1S 0
D  InpDate                       D

D AnySunday     C                  D'04/02/1995'
D WorkNum       S             7  0
D WorkDay       S             1S 0
```

Local data is very important for reusable procedure logic. It is also important to make maintenance tasks more reliable, as there is more independence within the subprocedures and, therefore, less chance that maintenance tasks will inadvertently affect mainline logic. In this example, the fields AnySunday, WorkNum and WorkDay are local fields to this subprocedure. They can only be referenced from the subprocedure.

We'll take a closer look at Local data later in the presentation.

Notes

Returning the result

Partner400

RETURN is used to send the result back to the caller

- It can simply return a value as in our original version

```
C                   Return    WorkDay
```

Or it can return an expression as shown below

Often a procedure will consist simply of a RETURN op-code

- For example, the entire logic for a procedure called CalcItemTotal which took the parameters Price, Discount and Quantity might be:
  - Return (Price * (Discount / 100)) * Quantity

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

Here we specify the return value for this subprocedure. We use the RETURN operation code and specify the return value in factor 2. Notice that the RETURN operation code is now a freeform operation.

The returned value can be either a single field or an expression, as in the second example. In fact, since a subprocedure can be used anywhere a variable of its type can be used, the returned value itself could be the result of a subprocedure invocation. But we're getting a little deep a little too quickly here ....

Notes

Defining the PRototype

Partner400

Each procedure requires a Prototype

- Notice that it's almost identical to the PI
- It must be present when compiling the procedure
  - This allows it to be validated against the Procedure Interface
- It is also required in each program that wants to use the procedure

The preferred approach is to code it as a /Copy member

- Place Prototypes for groups of related functions in a single member
  - Possibly one member per Service Program

Prototypes simply provide information to the compiler

- They don't result in any data definition

This is what our prototype currently looks like

```
D DayOfWeek       PR             1S 0
D  ADate                          D
```

The next step is to define the prototype. The parameters in the prototype must match the Procedure Interface (PI) because it also defines the interface to the procedure. The prototype will be used by the procedures that call this subprocedure.

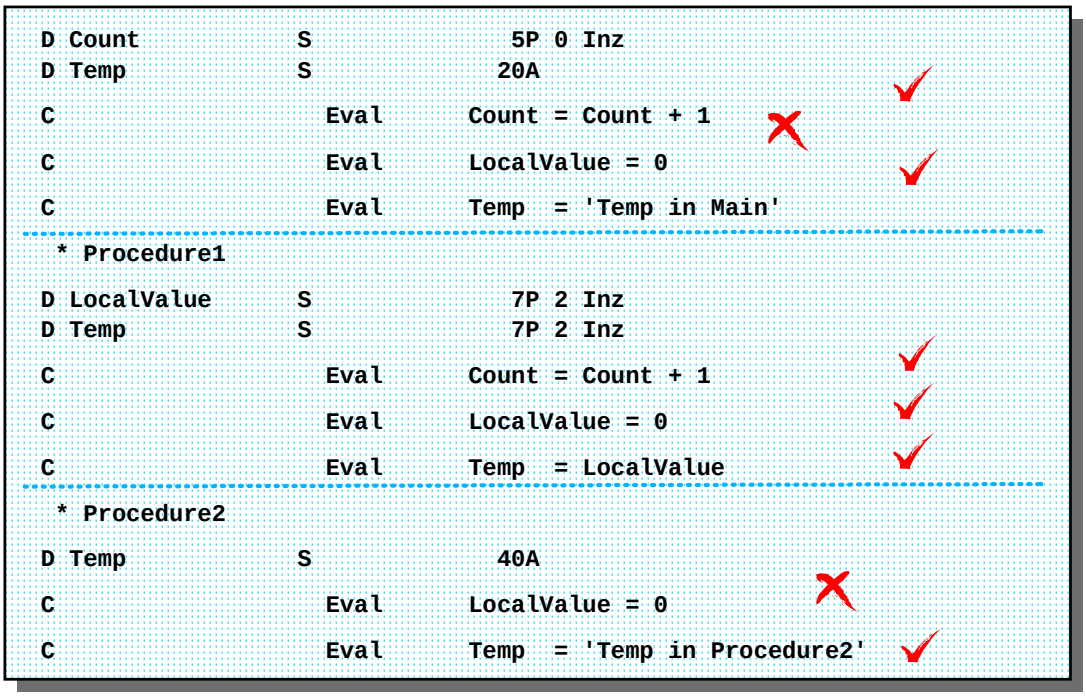
The prototype is also required to be placed into the module where the procedure is defined. This is so the compiler can check the validity of the prototype -- that is, that the parameters specified match the Procedure Interface in terms of number and type.

In the event that the subprocedure is placed in the same source member as the caller (as in our basic example), then only a single copy of the prototype is required, because the compiler will be able to check the prototype and procedure interface in a single compile step. If the subprocedure were to be placed in a separate source member, then a copy of the prototype would be required in both the member containing the subprocedure and in the main procedure (or calling procedure). as well as in any other main or subprocedures calling this subprocedure.

At this point in the development of the subprocedure, we are hard coding the prototype in the main line portion of the program. Once we have tested the subprocedure its prototype will be moved to a /COPY source member. This is a common (and encouraged) practice. The prototypes for subprocedures are grouped in a separate source member that is copied in (via the /COPY directive). This is especially important if the subprocedure is placed in a separate module (source member) because it is critical that the prototype in the calling procedure match the one in the defining procedure, since it is the once in the module containing the subprocedure that the compiler verified for you.

Notes

"Local" and "Global" Variables



Any and all subprocedures coded within the source member will automatically have access to global data items defined in the main procedure. They can also define their own local data fields, which will be accessible only within the subprocedure where they are defined.

As a rule of thumb, use of global data within a subprocedure should be avoided whenever possible. Ideally, subprocedures should act upon the data passed in through parameters and affect the data back in the calling code only by returning a result. This avoids the possibility of side-effects where (by accident or design) a subprocedure unexpectedly changes the content of a field back in the calling code.

In some circumstances accessing global data cannot be avoided. For example, although files can be used within a subprocedure, they can only be defined at the global level. Therefore in order to access the file data we must reference those global items.

In addition to returning values, subprocedures can also modify parameters passed to them, just as a parameter on a conventional program call can be modified. However, this is not the preferred approach, for the same basic reasons that we stated for global data items.

Notes

RPG IV Specification Sequence

H	Keyword NOMAIN must be used if there are no main line calculations.
F	File Specifications - always Global
D PR	Prototypes for all procedures used and/or defined in the source (Often present in the form of a /COPY)
D	Data definitions - GLOBAL
I	GLOBAL
C	Main calculations (Any subroutines are local)
O	GLOBAL
P ProcName1 B	Start of first procedure
D PI	Procedure Interface
D	Data definitions - LOCAL
C	Procedure calcs (Any subroutines are local)
P ProcName1 E	End of first procedure
P Proc..... B	Start of next procedure
.....	Procedure interface, D-specs, C-Specs, etc.
P Proc..... E	End of procedure
**	Compile time data

This chart illustrates the layout of a complete RPG program containing one or more subprocedures.

Note that the NOMAIN keyword on the H specification is optional and indicates that there is no mainline logic in this module, i.e., no C specs outside the subprocedure logic. Note also that any F specs always go at the top of the member, just after the H spec, for any files that will be accessed, either by the mainline code (if any) or by the subprocedures. This is true regardless of whether there is any mainline logic or not.

The first D specs are the PR(ototypes) for any subprocedures that are going to be used or defined in this source member. It is not compulsory to have them first, but since you will not need to reference or change them very often it is a good idea to have them near the top. Of course any prototypes for subprocedures that you are used in multiple programs should be in a /COPY member and not cloned from program to program.

The D and I specs that follow are for data items in the mainline, which are global, i.e., they can be accessed from both mainline logic and any subprocedures in this module.

Following the O specs for the mainline code is the beginning P spec for the first subprocedure. It is followed by the PI (procedure interface) for that subprocedure. D and C specs for this subprocedure are next, followed by the ending P spec.

Any other subprocedures would follow this, each with its own pair of beginning and ending P specs.

Procedures using Procedures

Partner400

How about a procedure to provide the day name ("Monday", "Tuesday", etc.) for a specified date ?

- This procedure will use DayOfWeek to obtain the day number

Most RPG programmers would code it this way

```
P DayName          B
D                  PI             9
D  InpDate                         D

D DayData          DS
D                                 63    Inz('Monday   Tuesday  Wednesday+
D                                       Thursday Friday   Saturday +
D                                       Sunday   ')
D DayArray                         9    Overlay(DayData) Dim(7)

D WorkDay          S               1  0 Inz

C                    Eval      WorkDay = DayOfWeek(InpDate)
C                    Return    DayArray(WorkDay)

P DayName          E
```

This chart illustrates how subprocedures can use other subprocedures.

In saying that most RPG programmers would tend to program it as shown is probably an overstatement. More likely is that they would tend to write the subprocedure to accept a day number (which of course they would obtain by using "DayOfWeek") and return the day name.

This is really "RPG/400 Think" though. Often we will require the name without needing to know the day number - so why have to call one routine just to pass the returned value to another! Instead we will pass the new subprocedure a date, and have it call the "old" one for us.

The new "DayName" procedure will call the "DayofWeek" procedure to get the number of the day of the week. The "DayName" procedure then translates the number into a day name.

Note that we must include AT LEAST 2 prototypes: one for DayName and one for DayofWeek, since DayName calls DayofWeek.

Notes

An alternative approach

There's nothing "wrong" with that approach but ....

- Earlier we said that a subprocedure can be used anywhere that a variable of its type can be used
- So we can replace these three lines of code:

```
D WorkDay         S              1  0 Inz

C                   Eval      WorkDay = DayOfWeek(InpDate)
C                   Return    DayArray(WorkDay)
```

With this more streamlined version

- Notice that we avoid the need to declare the 'WorkDay' variable !
  - If you find this hard to understand - do not attempt to learn Java!

```
C                   Return    DayArray(DayOfWeek(InpDate))
```

Since subprocedures which return a value can be used anywhere a field name or constant of that type (i.e., alphanumeric or numeric) and size can be used, the second example you see here could be used to incorporate a "cleaner" programming style. After all, why create a field to hold the day number simply to use it as a subscript and then throw it away!

Notes

Pause for Thought

Partner400

So far we've just been using procedures in the program

- A sort of "muscled up" subroutine

What if we want to reuse them easily ?

- The answer is Service Programs !!

But there are some changes needed

- We must separate the components:
  - The prototypes
  - The main program logic
  - And the subprocedures
- And add some additional keywords to the subprocedures

Since these two subprocedures might be very useful in many other programs that use dates, it would be a good idea to put this type of subprocedure into an ILE Service Program. That way, many programs can access one copy of these subprocedures.

On the following chart, we will see how the source needs to be "cut up" into the individual building blocks that will be used.

Notes

Separating the Components

Partner400

```
D DayOfWeek       PR             1S 0
D  ADate                          D
D DayName         PR             9
D  ADate                          D
:                 :               :
C                   Eval      DayNumber = DayofWeek(InputDate)
C                   Eval      TodayCh   = DayName(DateToday)
:                 :               :
P DayOfWeek       B
D DayOfWeek       PI             1S 0
D  InpDate                        D
C       :             :           :
C                   Return    WorkDay
P DayOfWeek       E
P DayName         B
D                 PI             9
D  InpDate                        D
        :             :           :
```

Prototypes

Main program

Subprocedures

This chart represents the source file as we now have it.

The prototypes for the two subprocedures are at the top, they are followed by the main processing logic that invokes the subprocedures. Last but not least is the logic for the subprocedures themselves. These prototypes will be "cut" from the source and placed in a separate source member. Later on we may add other prototypes for new date subprocedures (or indeed any new subprocedure).

In the source member that contains the main logic, we will need to add a /COPY directive to copy the prototypes into the source. Note "/COPY" NOT copy as in CC in SEU!!! Without the prototypes, the compiler will not know how to interpret the calls to DayName and DayOfWeek.

Now we must "cut" the source for the subprocedures and place them in their own source member as we will be compiling them separately to create the service program. Don't forget that we will also need to /COPY the prototypes into this source member as well. On the next chart we will look at the other changes that we will need to make to the subprocedures.

Notes

Creating the Subprocedures

- First copy the subprocedure logic into a new source member
- Next add an H-spec with the NOMAIN keyword
  - This makes the module "cycle-less"
- Now add the keyword EXPORT to the P-specs
  - This makes the procedures "visible" outside of the module
- Finally add the /COPY directive that will bring in the prototypes

```
H NOMAIN
 /Copy DateProtos

P DayOfWeek       B                   EXPORT
D DayOfWeek       PI             1S 0
D   WorkDate                      D
 *          Subprocedure logic is here
P DayOfWeek       E

P DayName         B                   EXPORT
D DayName         PI             9
D   WorkDate                      D
 *          Subprocedure logic is here
P DayName         E
```

The steps identified on this chart should be performed for any subprocedures that will be placed into a separate module from the program or procedure that calls them. The most common time to use this is when the subprocedures are to be placed in an ILE Service Program.

The NOMAIN keyword will provide much faster access to these subprocedures from other modules. It also tells the compiler NOT to include any RPG cycle logic in this module. The NOMAIN keyword is only allowed if/when there are no main line calculations in the source member PRIOR TO the first subprocedure. In other words, NOMAIN can only be used if there is no main procedure logic coded in this module.

EXPORT makes the name of the procedure "visible" outside of the Module. If it was not made visible in this way it could not be called by anyone outside of the module. For example, if DayOfWeek did not have the EXPORT keyword, it could still be called by DayName but not by any code outside of the module.

Notes

Changes to the main line code

Not much to do here

- Mostly just deleting stuff

- Start by copying the prototypes into a new source member
- Replace the prototypes in the main source with the necessary /COPY directive
  - This will be identical to the /COPY we placed in the subprocedure source
- Next remove all of the subprocedure logic form the main line program
- Note that you now have three source members
  - The main line program
  - The subprocedure source
  - And the prototype source

All that remains is to compile the pieces

```
 /Copy DateProtos
     :              :              :
C                   Eval      DayNumber = DayofWeek(InputDate)
C                   Eval      TodayCh   = DayName(DateToday)
     :              :              :
```

The NOMAIN keyword will provide much faster access to these subprocedures from other modules. It also tells the compiler NOT to include any RPG cycle logic in this module. The NOMAIN keyword is only allowed if/when there are no main line calculations in the source member PRIOR TO the first subprocedure. In other words, NOMAIN can only be used if there is no main procedure logic coded in this module.

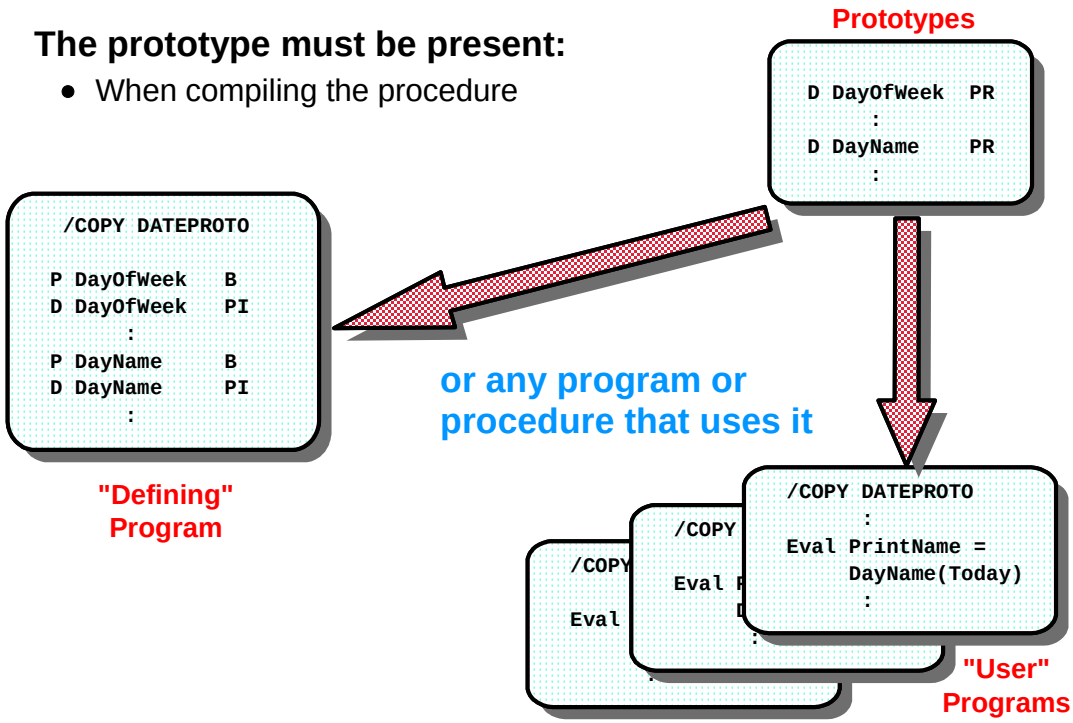
EXPORT makes the name of the procedure "visible" outside of the Module. If it was not made visible in this way it could not be called by anyone outside of the module. For example, if DayOfWeek did not have the EXPORT keyword, it could still be called by DayName but not by any code outside of the module.

When compiling remember that you don't need to compile the prototypes by themselves (in fact they won't compile!). They will be processed by the compiler when it encounters the /COPY directives in the other sources. Also remember not to include H specs in your prototype source member - they will almost certainly result in an "out of sequence" error when incorporated into the main line or subprocedure logic. (The exception to this rule is when you are using conditional compiler directives to control what gets copied - but that topic is outside the scope of this session)

If you don't want to create a Service Program for your early subprocedure experiments, then you can simply compile both the main line and the subprocedures using CRTRPGMOD (PDM option 15). Then when both modules have been compiled, use CRTPGM to "bind" them together.

Notes

Reminder on Prototype Usage



One of the most common mistakes made when programmers first begin using subprocedures and prototypes is to fail to include prototypes in all the places where they are needed.

This chart tries to make the rules more clear. These three blocks represent the three components that we just split our program into. i.e. The prototypes, main program, and subprocedures. The term "Defining Program" on this chart means the module (source member) where the logic of the subprocedure is coded. The "User Program(s)" are any modules (source members) that will call or refer to these subprocedures.

The prototypes must be present in the source member where the RPG IV subprocedures are DEFINED and also in every source member where the subprocedures will be USED. This is so that the compiler can check the prototype against the Procedure Interface (PI). This rule may seem silly to you, but the compiler enforces it so .....

The fact that these prototypes are required in multiple places is the reason we strongly recommend that the /COPY directive be used to bring in the prototype code. This way, you can ensure the correct prototype is always used.

Notes

Enhancing the Date Routines

Currently our date routines only work with a single format

- i.e. The default at the time of compilation (probably *ISO).

The CONST keyword can help us here

- It allows the compiler to accept a date in any format
  - Remember: Both the prototype & the procedure interface must change
- Use of the DATFMT keyword is also recommended
  - This avoids the possibility of the procedure being compiled with a different default date format to the one in force when the function is used.

```
 * Prototype for DayOfWeek procedure
D DayOfWeek           PR             1  0
D InputDate                           D   CONST DATFMT(*USA)
 * Prototype for DayName procedure
D DayName             PR             9
D WorkDate                            D   CONST DATFMT(*USA)
```

The DATFMT keyword should always be specified for any date fields passed as parameters or returned by a subprocedure (in other words any date field defined within a Procedure Interface (PI) or Prototype (PR). Without it, the format of the date is open to interpretation at the time of compilation. Without the DATFMT keyword, the date will be classified as the default type for the compilation. While this is normally *ISO, it is subject to change at the whim of an H-spec DATFMT entry. The result could be that within the subprocedure the date is viewed as *ISO while in the calling program it is defaulting to *USA. Not a recipe for success!!

In this example, by combining CONST and the DATFMT keywords, the compiler can generate a temporary (hidden) date field in the calling program or procedure, if necessary, in order to convert the date format used by the caller to the format (in this case, *USA) used in the called subprocedure.

In general, the use of the CONST 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 it is acceptable that the compiler to accommodate such mismatches by making a copy of the parameter (if necessary) prior to passing it.

The actual parameter is still passed to the callee in the same way i.e. a pointer to the data is passed. This differs from the keyword VALUE, which in other respects has a similar effect to CONST as you will see on the next chart.

Notes

Passing parameters by VALUE

Passing parameters by VALUE is an alternative approach

- Like CONST it allows the compiler to accommodate mismatches
- Unlike CONST the actual data is passed to the called procedure
  - Not a pointer to the data
- Since it is a copy of the data the callee is allowed to make changes

VALUE is only permitted for bound calls

- i.e. It cannot be used when calling a *PGM object

```
 * Modified Prototype for DayOfWeek procedure
D DayOfWeek           PR             1S 0
D                                     D   VALUE DATFMT(*USA)
```

```
 * Modified Procedure Interface for DayOfWeek procedure
D DayOfWeek           PI             1S 0
D WorkDate                            D   VALUE DATFMT(*USA)
```

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.

Since AS/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 CALLP/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 and conversions needed before passing the parameter.

Notes

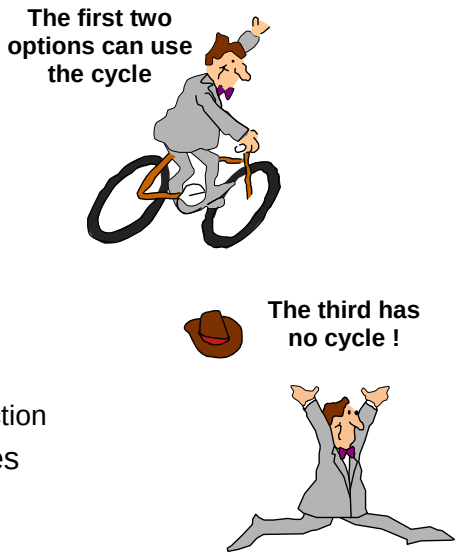
Types of RPG IV "Programs"

RPG III style

- Main line code
- No subprocedures

RPG IV style

- Main section
  - Defines files and variables
    - Both are Global
  - C specs define mainline logic
- Local Subprocedures
  - May define their own local variables
  - And access those defined in the Main section
- May also access external Subprocedures



RPG IV - Subprocedures only

- Main section can define files and variables but No logic
- These subprocedures are almost as fast to execute as an EXSR !!

When using subprocedures, you can write the subprocedure code in the same source member as the mainline code that will call it. Alternatively, you can place your subprocedures in a separate module, preferably using the "cycle-less" support mentioned on the chart. Or you can combine the two, writing local subprocedures that will only be sued by the mainline code, while also calling common routines that have been compiled independently.

When you do include subprocedures in the same source member as the mainline, then you would have a single main procedure, followed by one or more subprocedures. The main procedure defines all files and global data. Global data items are usable by any subprocedures coded in the same module (although as we noted earlier this is not really a good idea). Fields in files are always considered global, as all files must be defined in the main procedure and data defined in the main procedure are global within the module.

RPG IV subprocedures can be made "cycle-less". That means that the RPG cycle is not generated by the compiler, as it is for all RPG IV modules with a mainline. The benefit of this is that calls to subprocedures in these cycle-less modules are very fast -- close to the speed of EXSR operations. If you want subprocedures of this type to access files, you must declare them in the global part of the source - just don't define any main line logic. Note that if you use files in a NOMAIN source the compiler will issue a warning that you should explicitly open/close the files. While this is a good idea it is not essential and you can ignore the warning during your early explorations.

When you first try to create subprocedures you will probably meet the compiler message "RNF3788 - Keyword EXTPGM must be specified when DFTACTGRP(*YES) is specified on the CRTBNDRPG command". This occurs because the default values on the CRTBNDRPG command assume that the compilation is for an OPM style program (sometimes known as compatibility mode). Subprocedures are an integral part of the ILE environment and, if you are using them you are therefore using features of ILE, so the default value will not suffice. To avoid this message, you must remember to compile with the option DFTACTGRP(*NO). Better still - embed it on an H-spec in the source so you can't forget.