RPG IV Subprocedures Basics

Jon Paris
Jon.Paris@Partner400.com
www.Partner400.com

Partner400
Your Partner in AS/400 and iSeries Education

What is a Subprocedure ?

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

They can be used in the same way as IBM's Built-In-Functions

- Except we don't get to put a cute little "%" sign in front of the name!

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

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 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.

Later we will build the subprocedure DayOfWeek

- It takes a date as a parameter
- And returns the single digit day number
  - In our case, 1 = Monday; 2 = Tuesday; etc.
- So we can use it anywhere that we could use a one digit numeric
  - See the examples below

```
C           If        DayOfWeek(ADateFld) > 5

C           Eval      WeekDay = DayOfWeek(Today)

C           Eval      DayName = NameArray( DayOfWeek(AnotherDate) )
```

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".

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                   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!

Basic Subprocedure

```
D DayOfWeek       PR             1S 0                    5
D  ADate                          D
:
C                   Eval      DayNumber = DayofWeek(InputDate)    1
:                    :         :
P DayOfWeek       B                                      2
D DayOfWeek       PI             1S 0
D  InpDate                        D
D AnySunday       C                    D'04/02/1995'              3
D WorkNum         S              7  0
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 and it's invocation from the main line of the program. The "boxed" numbers on the chart correspond with numbers on the following charts which discuss each component in more detail.

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.

Invoking the Subprocedure

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.

The complete code is shown here so that you can see the code in context.

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

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
P DayOfWeek       E
```

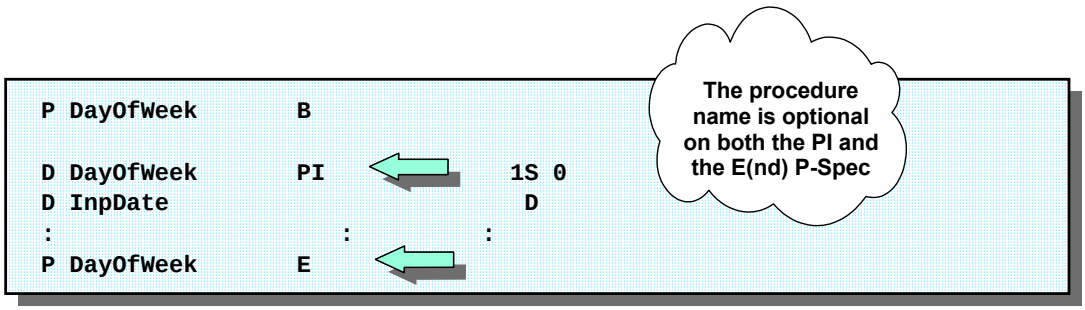
2 P-specs and the PI

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 (PI) is the equivalent to the *ENTRY PLIST- it defines the parameters passed to the subprocedure. The PI is typically the first D specs in the subprocedure. The length and type definitions (1S 0) appearing on the same line as the PI specifies the format of the return value. Note: It is possible to have a subprocedure that returns NO value in which case the entry is left blank.

The data item(s) that follow the PI (with blanks in the PI columns) are the parameters to the subprocedure. i.e they are equivalent to the PARM entries in an *Entry PLIST. In this example the ADate field is the only parameter passed.

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

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
P DayOfWeek       E
```

3 DayOfWeek Subprocedure

Subprocedures allow us to define local variables

- They can ONLY be referenced within the subprocedure

Much safer than the traditional RPG subroutine approach

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

Note that the field 'WorkNum' terminates the PI

- Just as it would terminate a Data Structure

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

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

Local data is very important for reusable procedure logic. It is also important to make maintenance tasks more reliable, as there is 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.

Again the complete code is shown here so that you can see the code in context.

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

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
P DayOfWeek       E
```

4 Returning the result

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

- Note that having multiple Return operations is generally frowned upon, we have done it here just to demonstrate the possibility

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

- We will see an example later

```
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 ..

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

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                    <<<<<<<<<<<<<<<<<<
P DayOfWeek       E
```

5 Defining the Prototype

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

For subprocedures that will be used by multiple programs, 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.

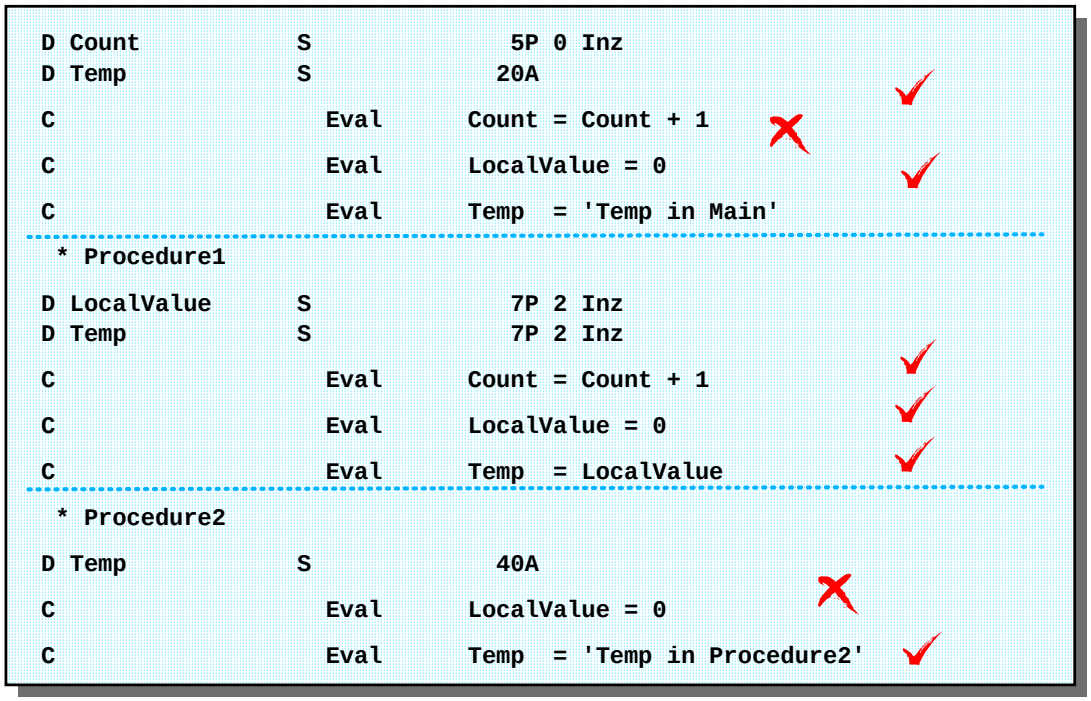
In the event that the subprocedure is placed in the same source member as the caller (as in our basic example), then only a single copy of the prototype is required, because the compiler will be able to check the prototype and procedure interface in a single compile step. If the subprocedure were to 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.

Not enough room here for all the code - just a reminder of the position of the PR in the source:

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

"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 discussed for global data items.

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

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.

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!

Pause for Thought

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.

Separating the Components


```
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.

Start by "cutting" the prototypes from the source and place them in a separate source member. Later on we may add other prototypes for new date subprocedures (or indeed any new subprocedure). Do not make the mistake of including H specs in the prototype! Remember the prototype type source will never be compiled on its own. It is compiled by being /COPY'd into the source of the subprocedures and any other programs that wish to use the subprocedures.

Note that whenever we use the term "/COPY" we mean the compiler directive /COPY and NOT copy as in CC in SEU!!!

Next "cut out" 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 add a statement 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.

What we are left with will be the source for the main (using) program. We will of course need to add a /COPY directive to bring the prototypes into the source. Without those prototypes, the compiler will not know how to interpret the calls to DayName and DayOfWeek.

Note: Building and using Service Programs is outside the scope of this session, but we have provided brief instructions on each of the next two pages. E-mail us for help if you still encounter problems.

Creating the Subprocedures

- Copy the subprocedure logic into a new source member
- Add an H-spec with the NOMAIN keyword
  - This will make the resulting module "cycle-less"
- 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's logic is here
P DayOfWeek       E
P DayName         B                   EXPORT
D DayName         PI             9
D   WorkDate                      D
 *          Subprocedure's 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.

To compile the subprocedure source:

We will compile using the CRTRPGMOD command, this will create a *MODULE object that we can then either build into a Service Program, or bind directly to the main module.

If you are using PDM, option 15 will run the CRTRPGMOD command for you. Don't forget that if you want to use the debugger on the resulting objects you must request that now. We recommend that you use the *LIST (or *ALL) options.

If you choose to create a Service Program (and after all that is what we are discussing here) you will next use the CRTSRVPGM command. You can give the Service Program its' own name, or name it after the module. For simplicity you should also specify the option *ALL for the "Export" parameter. Without this the names that you exported from the module will not be exported by the service program.

Changes to the main line code

Not much to do here (mostly just deleting stuff)

- Delete the prototype entries in the source member
- Add a /COPY directive in their place
  - This will be identical to the /COPY we placed in the subprocedure source
- Next delete all of the subprocedure logic
  - We will be "binding" this program to the logic in the Service 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)
    :                 :                 :
```

Just a reminder 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)

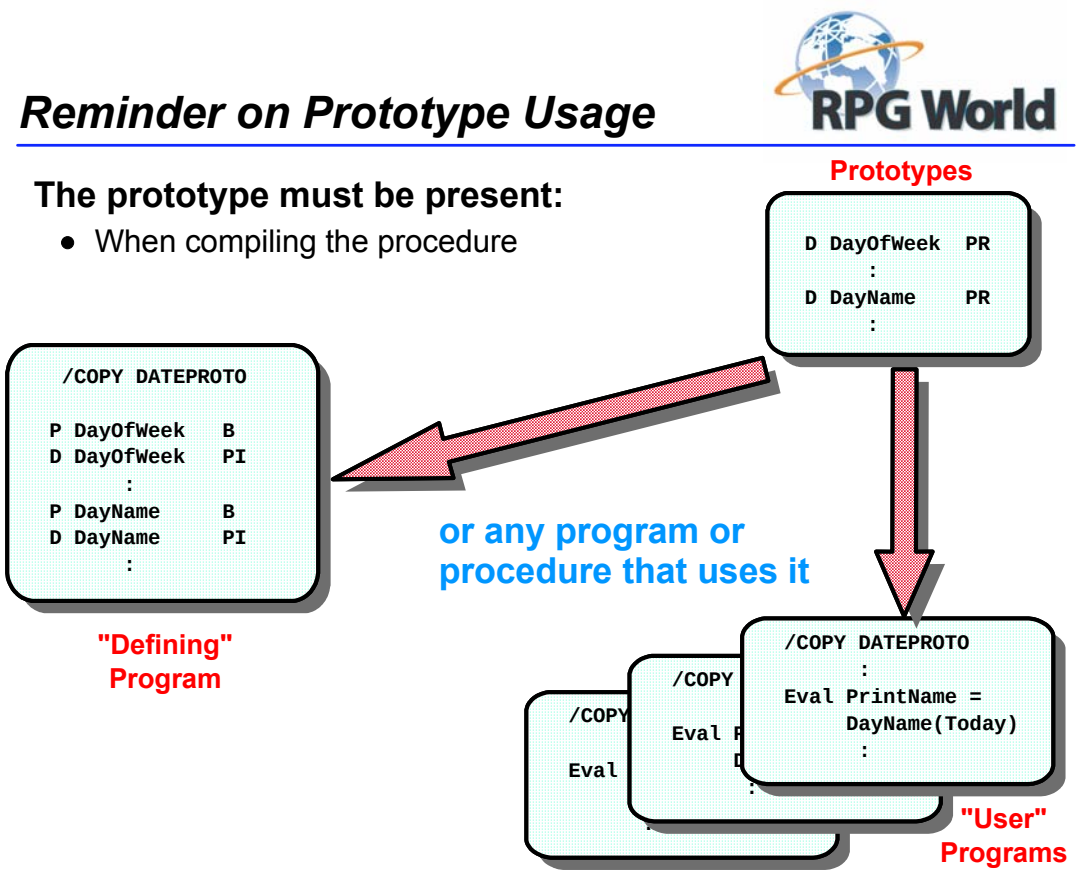
Creating the Main Program and linking to the Service Program:

Once again we need to start by creating a *MODULE object using CRTRPGMOD (PDM option 15). There is a way to use CRTBNDRPG (option 14) which we will mention briefly in a moment.

Once the module has been created, we need to link it to our Service Program to create the *PGM object. To do this use the CRTPGM command. For simplicity specify the name of your module as the program name, and the name of the Service Program you just created as the "Bind Service Program" entry. Note that you will not see this parameter unless you press F10 to list all parameters, and then scroll to the second page - it is the first entry on that page. Further down on that same page note that Activation Group is specified as *NEW. This is OK while you are "playing" but not a good choice in your production code most of the time. Sorry we don't have space to explain why here - check out one of the ILE sessions for details.

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 specifying the names of both modules (the main program should be specified first) to "bind" them together. This will create a program that is "Bound by copy". Again this is fine for experimenting with subprocedures, but not a good way to go for production code.

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.

Problems with Dates

Our date routines are now compiled separately

- But we didn't specify a format for the date field
- Q: So what format is it in?
  - A: The format that was the default at the time of compilation
- Q: Does that present a problem?
  - A: Quite probably - unless we only ever use *ISO

So ALWAYS specify a format

- When using dates as parms OR return values

```
 * Prototype for DayOfWeek procedure
D DayOfWeek            PR             1  0
D InputDate                            D      DATFMT(*USA)

 * Prototype for DayName procedure
D DayName              PR             9
D WorkDate                             D      DATFMT(*USA)
```

The DATFMT keyword should always be specified for a date field returned by a subprocedure, or passed as a parameter. In other words any date field defined within a Procedure Interface (PI) or Prototype (PR) should have the DATFMT keyword.

Why? Well without it, the format of the date is determined 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.

Suppose that our prototype does not specify the date format. When we /COPY it into a program with no H-spec, the compiler will interpret it as an *ISO date. If the program then passes an *ISO date to the subprocedure, the compiler will accept this as valid.

Now suppose that in the source member that contains our subprocedure, we have an H-spec that specifies the default format as being *USA. The same /COPY member will now have it's date fields interpreted as being *USA dates.

The result would be that within the subprocedure the date is viewed as *USA while in the calling program it is defaulting to *ISO. The prototype that should protect us from such problems is allowing a date of the wrong format to be passed. Not a recipe for success!!

Note that although we have emphasized this as being an issue with dates, the same holds true for Times. It is not and issue with Timestamps since they only have one format.

CONST (Read-only) Parameters

There's one problem with specifying DATFMT

- The routines now only work with parameters in that format
- The compiler will reject any attempt use any other format

The CONST keyword can help us here

- It allows the compiler to accept a date in any format
  - The compiler generates a temporary field in the correct format and moves the parameter into it before passing the copy
- Remember: Both the prototype & the procedure interface must change

```
 * 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)
```

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.

Note that CONST indicates that the parameter is Read-only i.e. the called subprocedure cannot change it's value. The compiler will in fact defend against this and will error out any attempts to change the value of parms passed as CONST.

There is no equivalent to CONST needed on the definition of the returned value. If the compiler notes that (for example) and *ISO date is being returned and it being assigned to a *USA date field, it will automatically convert the date just as it would in a simple assignment of one date type to another.