Code Procedure Prototypes with the Procedure

Craig Pelkie
Bits & Bytes Programming, Inc.
craig@web400.com
Code Procedure Prototypes with the Procedure
Edition PROCPROTO_20020219

Published by

Bits & Bytes Programming, Inc.
Valley Center, CA 92082
craig@web400.com

Copyright © 2002, Craig Pelkie. All Rights Reserved

You are allowed to print copies of this document for your own personal use and for the use of others in your company or organization. Otherwise, no part of this document or any accompanying computer source code may be reproduced or distributed in any form or by any means, or stored in a database or data retrieval system, without the prior written permission of Craig Pelkie, who is the author of the document and the computer source code.

All computer source code distributed with this document, either on diskettes, CD-ROM, or available for downloading from sources such as the Internet is Copyright © 2002 Craig Pelkie, All Rights Reserved. The source code is for use in computer programs that you develop for internal use within your company, or for use within programs that you develop for the use of your clients. No part of the computer source code distributed with this document shall be reproduced in source code format, either printed or in electronic format, by you or by others who you allow to have access to the source code. You shall not cause the source code to be stored on any information retrieval system, such as the Internet. You shall not develop any written articles, books, seminar materials, or other presentations that include the source code provided on the diskettes accompanying this document or within the document itself.

For any questions regarding your rights and responsibilities using the computer source code distributed with this document, contact Craig Pelkie, Bits & Bytes Programming, Inc., who is the owner of the source code.

LIMITATION OF LIABILITY AND DISCLAIMER OF WARRANTY

No representation is made that any of the techniques, programs, computer source code, commands, or configurations described and depicted in this document and on the computer source code accompanying this document are error-free and suitable for any application that you may develop. Craig Pelkie makes no warranty of any kind, expressed or implied, including the warranties of merchantability or fitness for a particular purpose, with regard to the information, examples, and computer source code presented in this document and on the accompanying media. Everything provided in this document and the accompanying computer source code, if any, is provided “as is”. Craig Pelkie shall not be liable in any event for incidental or consequential damages or any other claims, pursuant to your use of any of the techniques presented in this document, or your use of the computer source code, even if Craig Pelkie has been advised of the possibility of such damages.

You are responsible for testing any and all programs, configurations, commands, and procedures described in this document prior to using the programs, configurations, commands, and procedures with important user data. You must ensure that adequate and sufficient backup of important user data is available, in the event that recovery of the important user data is required.
Code Procedure Prototypes with the Procedure

Note: I am indebted to Bryan Meyers, the popular News/400 RPG columnist, for describing this technique to me several years ago.

One of the first things you notice as you start creating procedures instead of subroutines is that there is quite a bit of setup work to do. In addition to the code in the procedure itself, you need to create a procedure prototype and a procedure interface. The procedure prototype is used to “advise” the compiler what it should expect when it encounters your use of a procedure in your code. The procedure interface is a parameter list for the procedure itself, used to receive values and optionally return one or more values to the caller.

A few years ago, I was working with a programming team to develop a system that made extensive use of MQ Series in ILE RPG programs. We decided to use procedures and service programs to extract commonly used and useful code. Before adopting the technique I describe in this article, we spent quite a bit of time trying to keep the prototypes and interfaces “in sync”. Because the procedures within the service program might call each other, we had to code prototypes in the service program in addition to the procedures. To use the procedures in the service program from an ILE RPG program, we copied the prototypes into the ILE RPG source code.

As with any copy-and-paste technique, it quickly became tedious to keep the prototypes and interfaces synchronized, especially in the many ILE RPG programs using the service program. Because the system we were developing was constantly changing, the prototypes and interfaces were also changing. Keeping everything together became more difficult as we added procedures to the service program.

The programming team decided that we could make good use of the /COPY compiler directive, and put procedure prototypes in separate source members. For example, the following code is the prototype for procedure getMqmQueueName:

```
D getMqmQueueName...
D                 pr                  like(queueName)
D appPlist                            value
D                                     like(appParms)
D qID                                 const
D                                     like(queueID)
```

Two parameters are passed to the getMqmQueueName procedure and it returns a single parameter value. The parameters and return value are defined based on a standard set of field definitions using the like keyword. The advantage of that technique is that we could easily change the parameter definitions if necessary. For example, the queueID is defined in another /COPY module as a CHAR(2) field and queueName as a CHAR(48) field (MQ Series allows queue names of that length). If we needed to change the lengths of either field, we simply made the change in the definition module and recompiled; the like keyword picks up the new definition so that we don’t have to go back and manually change all of the prototypes.

The corresponding procedure interface for the getMqmQueueName procedure looks like this:

```
D getMqmQueueName...
D                 pi                  like(queueName)
D appPlist                            value
D                                     like(appParms)
D qID                                 const
D                                     like(queueID)
```

The two code fragments are obviously almost identical. Again, the problem arose when we needed to change the procedure interface to modify the functionality of the procedure. We then needed to locate the corresponding /COPY module containing the procedure prototype and update it, then recompile all of the programs using the module.
**DEFINE to the rescue**

The technique that we adopted to get control over the changes uses the /DEFINE and /UNDEFINE compiler directives. Using those directives, we could now create a procedure module like this:

```
/IF DEFINE(PROCHEADER)
D getMqmQueueName...
D     pr       like(queueName)
/ENDIF

/IF DEFINE(PROCBODY)
P getMqmQueueName...
P     b       export

D getMqmQueueName...
D     pi       like(queueName)
/ENDIF

D appPlist       value
D   like(appParms)
D qID    const
D   like(queueID)
/IF DEFINE(PROCBODY)

**************************************************************************
* Instance data for procedure
*
* dtaData       - data retrieved from data area
* errorOccurred - error flag
* msgID         - message ID of error message to retrieve
* qName         - MQM queue name
**************************************************************************

D dtaData         s       like(dataAreaData)
D errorOccurred   s       like(flagField)
D msgID           s       like(systemMsgID)
D qName           s       like(queueName)

**************************************************************************
* Code for the procedure - not shown in this example
**************************************************************************

C       return    qName

P getMqmQueueName...
P     e
/ENDIF
```

The getMqmQueueName procedure is now coded in a separate source member named GETMQMQNM, which we kept in a source file named QPROCEDURE. For now, notice the following:

- Statements to define the procedure prototype are in the /IF DEFINE(PROCHEADER) block.
- Statements to define the procedure interface are in the /IF DEFINE(PROCBODY) block (the first of two such blocks).
- The procedure parameters are defined immediately following the first /IF DEFINE(PROCBODY) block. The parameters are not within a DEFINE block.
The compiler includes or excludes code depending upon the current conditions that are defined. You’ll see shortly where we defined the conditions. For now, understand that if the PROCHEDER condition is defined, the code in its IF block is included in the source code that is passed to the compiler. If the PROCHEDER condition is not defined, the code is not passed through to the compiler. The same rules apply to the PROCBODY define.

To help you visualize the end result, look back on the first page at the procedure prototype and the procedure interface, then trace through the statements in the getMqmQueueName procedure. You should be able to see that if you can control when PROCHEDER and PROCBODY are defined, you can select either of the two resulting sets of statements.

**The Service Program source code**

The PROCHEDER and PROCBODY conditions are actually defined in the source code for the service program. The complete code for the service program is this:

```
H nonain debug
**************************************************************************
* MQMPROCS - This module contains commonly used service
* procedures for MQM processing.
**************************************************************************

**************************************************************************
* Copy procedure headers
**************************************************************************
/DEFINE PROCHEDER
/COPY #MQMPROCS MQM service procedures
/COPY #SYSPROCS System service procedures
/UNDEFINE PROCHEDER

**************************************************************************
* Copy procedure bodies
**************************************************************************
/DEFINE PROCBODY
/COPY #MQMPROCS MQM service procedures
/UNDEFINE PROCBODY
```

This source member is named MQMPROCS and was in our QRPGLESRC source file. To compile the service program, we used the CRTRPGMOD command followed by CRTSRVPGM.

In the upper set of statements, the PROCHEDER condition is defined. Note that a /DEFINE directive does nothing more than identify a condition. You can name a condition almost anything you want. You do not assign a value to the condition, you simply /DEFINE or /UNDEFINE it, as shown in the directives. Although the condition is not “used” in this source member, its definition is passed through to the other source code that is included in the member via the /COPY directives.

The PROCHEDER block is used to enable the procedure prototypes. Note that within the scope of the PROCHEDER define there are two /COPY modules. The #MQMPROCS copy includes procedure prototype source code for the MQ Series procedures, including the getMqmQueueName procedure. The #SYSPROCS copy includes the prototype source code for procedures in another service program.

Following the two /COPY directives, the /UNDEFINE PROCHEDER directive causes the condition to be undefined. Again, although the condition is not “used” in this source code, the fact that it is now undefined is “known” to the code that is copied in at compile time.

Now that the procedure headers have been processed, we need to include the procedures in the source code passed to the compiler. The procedures include the procedure interface plus the code that does the actual work in the procedure. The /DEFINE PROCBODY directive is used to indicate that we now want to include that code. Although it seems counterintuitive, we use the /COPY #MQMPROCS directive again. At this point, code conditioned on the PROCHEDER condition is not included.
The #MQMPROCS source code

The final source code to look at is the “intermediate” source code, the #MQMPROCS member that is used on the /COPY directive in the MQMPROCS member. The complete code for the #MQMPROCS member is as follows:

*****************************************************************
* #MQMPROCS - Copy procedure headers (interface definitions)
*****************************************************************

/IF DEFINED( PROCHEDER)

/COPY QPROCEDURE, CONNECTMQM                               connectMqm
/COPY QPROCEDURE, FMTMQMAUD                                formatMqmAuditMsg
/COPY QPROCEDURE, GETMQMAUD                                 getMqmAuditMsgID
/COPY QPROCEDURE, GETMQM OPT                                getMqMnputOptions
/COPY QPROCEDURE, GETMQM ET                                 getMqJournalEntryType
/COPY QPROCEDURE, GETMQMMSG                                 getMqmMsg
/COPY QPROCEDURE, GETMQMDOPT                                getMqmOutputOptions
/COPY QPROCEDURE, GETMQMNM                                  getMqmQueueManagerName
/COPY QPROCEDURE, GETMQMN                                  getMqmQueueName
/COPY QPROCEDURE, HNDMQMERR                                 handleMqmError
/COPY QPROCEDURE, LOGMQMVT                                  logMqmEvent
/COPY QPROCEDURE, OPNMQMNI D                                openMqmInputQueueByID
/COPY QPROCEDURE, OPNMQMMN                                  openMqmInputQueueByName
/COPY QPROCEDURE, OPNMQMOUID                                openMqmOutputQueueByID
/COPY QPROCEDURE, OPNMQMOUNM                                openMqmOutputQueueByName
/COPY QPROCEDURE, OPNMQM                                    openMqmQueue
/COPY QPROCEDURE, PRPMQM                                   prepareMqmQueue
/COPY QPROCEDURE, PUTMQMMSG                                 putMqmMsg

/ENDIF

*****************************************************************
* Copy procedure bodies
*****************************************************************

/IF DEFINED( PROCBODY)

/COPY QPROCEDURE, CONNECTMQM                               connectMqm
/COPY QPROCEDURE, FMTMQMAUD                                formatMqmAuditMsg
/COPY QPROCEDURE, GETMQMAUD                                 getMqmAuditMsgID
/COPY QPROCEDURE, GETMQM OPT                                getMqMnputOptions
/COPY QPROCEDURE, GETMQM ET                                 getMqJournalEntryType
/COPY QPROCEDURE, GETMQMMSG                                 getMqmMsg
/COPY QPROCEDURE, GETMQMDOPT                                getMqmOutputOptions
/COPY QPROCEDURE, GETMQMNM                                  getMqmQueueManagerName
/COPY QPROCEDURE, GETMQMN                                  getMqmQueueName
/COPY QPROCEDURE, HNDMQMERR                                 handleMqmError
/COPY QPROCEDURE, LOGMQMVT                                  logMqmEvent
/COPY QPROCEDURE, OPNMQMNI D                                openMqmInputQueueByID
/COPY QPROCEDURE, OPNMQMMN                                  openMqmInputQueueByName
/COPY QPROCEDURE, OPNMQMOUID                                openMqmOutputQueueByID
/COPY QPROCEDURE, OPNMQMOUNM                                openMqmOutputQueueByName
/COPY QPROCEDURE, OPNMQM                                    openMqmQueue
/COPY QPROCEDURE, PRPMQM                                   prepareMqmQueue
/COPY QPROCEDURE, PUTMQMMSG                                 putMqmMsg

/ENDIF
Note that this member is also divided into two sections: the statements that are conditioned on the PROCHEADER condition and statements conditioned on the PROCBODY condition.

First, it must be stated that the #MQMPROCS member is not strictly necessary. We could have achieved the same effect by simply including all of its /COPY directives in the MQMPROCS service program source code. However, we felt that this "three-level" hierarchy better reflected our understanding of how the individual procedures fit into the service programs.

Looking at the #MQMPROCS source, it is pretty obvious that there are many procedures that we developed to work with MQ Series. By putting the list of procedures that are specific to the MQ Series functionality in this intermediate source member, we can easily add a new procedure to the service program, by simply adding the /COPY directive in the two conditional blocks and recompiling the module and service program. The values of the PROCHEADER and PROCBODY conditions are passed through to the actual procedure source code itself when compiled.

The alternate approach, of simply putting all of the /COPY directives into the MQMPROCS member, was not used because we wanted to make it obvious what was actually in the service program itself. If you look back at the MQMPROCS source code, you can easily see that the #MQMPROCS and #SYSPROCS "sets" are used. Because we also developed other high-level sets of procedures as service programs, we wanted it to be obvious from the source code which other sets were used in any of the service programs. Again, the hierarchy could have been just two levels, but we opted for the three level approach shown here.

**So what was the advantage again?**

I admit, it certainly looks like a lot more work to do this than to just create a source member for each procedure prototype, and copy it in as needed. But now look back again at the actual source code for the procedure on page 2. *All* of the code for the procedure is in one place. Once you have the PROCHEADER and PROCBODY blocks coded in the procedure itself, you can freely add to or change the procedure parameters; they are only defined in one place. Note that the procedure parameters are not defined within the scope of either PROCHEADER or PROCBODY, as the parameters are used by both the header (the prototype) and the body (the interface).

Another benefit we saw was that it was now very easy to have different programmers develop the individual modules in the service program. Rather than have all of the source code in one large source member, the modules are in separate source files and brought together at compile time. If all of the code for the service program was in one source member, that source member would be locked to just one programmer. Using our technique, each programmer could freely work on their procedure independently of each other.

By setting up the intermediate #MQMPROCS member, it is also obvious how a new procedure is added to the service program: simply open the member and add the two /COPY directives. To complete the process, one person was assigned as the "compiler", and would periodically recompile the module and the service program. The build steps would usually complete in a matter of minutes.

By splitting up the code into /COPY modules, it was also easy to isolate modules that contained compile errors. If the service program build failed, we could simply locate the failing procedure, comment out its /COPY directives in #MQMPROCS and send it back to the programmer.

Even if you are developing a service program by yourself, you might want to adopt some of the techniques described. I found it advantageous to keep procedures in individual source members, rather than put everything into what would have been a rather large single source member.

**Summary**

Admittedly, the technique I’ve described takes some getting used to. I’m sure that some of our team (there were six of us) were never entirely sure how it worked, but they were able to easily create new procedures and add them to the builds. Even though the compiler forces you to declare the prototype and then the interface, you are not required to put the code in two different places. You simply need to know how to use the additional tools provided by the compiler so that it will use the right code at the right time.