Working with APIs – Part 1

Working with APIs, Part 1

Two simple ILE RPG programs show how calling APIs using PLIST and prototypes differs

by Paul Morris

AS A FREELANCE ANAlyst/programmer, I've met many programmers who have widely different skills and experience. In working with these programmers, I've found that many of them not only lack awareness of what APIs can do, but they also have a fear of programming with them. The purpose of this article is to show how easy it is to use APIs to help overcome that fear.

Here, we examine two versions of a simple ILE RPG program that calls an API to see how the two programs differ. The first program calls an API with a PLIST, and the second one calls an API using prototyped calls. In Part 2 of this article, we'll build a program for a utility that compares the source timestamp on program objects with that on the source file and reports differences.

Calling a Program with PLIST

Figure 1 is a program written in a traditional style with a program call that uses a PLIST. This example employs an API to retrieve an object description, in this case, of a library. The program is called from the command line and is passed the library name as a parameter. If the library exists, the program returns normally; if the library doesn't exist, the program dumps before failing.

You can also easily step through this program using the STRDBG (Start Debug) command, which lets you see the variables changing. (For more information about using the ILE debugger, see "Using IBM's ILE Debugger, Part 1," April 2000, *www.iSeriesNetwork.com*, article ID 6707, and "Using IBM's ILE Debugger, Part 2" July 2000, article ID 7520.) Now, let's examine this program in more detail.

The copy member at A is a system include file of a data structure used to control errors. If you don't have the system include files on your system, you can load them from the installation CDs (more about error handling in a moment).

At B is another copy member that has the data structure to which the API returns data. In fact, more than one data structure is defined in the copy member, each showing more data than the first. The purpose of this is to enable different data to be returned at different costs (processing overhead). You choose the data structure you want when you call the API. (For a list of APIs and their functions, visit IBM's Info Center at *http://publib.boulder.ibm.com/iseries/v5r2/ic2924/info/apis/api.htm*.)

The API uses the work fields at C. The *ENTRY PLIST (at D) passes the name of the library to the program. The *INZSR subroutine at E initializes the error structure so that a simple RESET restores the values.

Because this program is the foundation on which we build the rest of the

FIGURE 1
Calling a program with PLIST

```
H debug
 *====================================================================
 * API copy sources
 *--------------------------------------------------------------------
A* API error data structure
 /copy QSYSINC/QRPGLESRC,QUSEC
 *--------------------------------------------------------------------
B* API structure for retrieving object description API
 /copy QSYSINC/QRPGLESRC,QUSROBJD
 *====================================================================
C* API parameter fields
d @Length         s                9b00
d @Format         s                   8
d @FilLib         s                  20
d @ObjTyp         s                  10
d @InLib          s                  10
 *====================================================================
 * Definitions
 *--------------------------------------------------------------------
D c      *Entry        Plist
 c                     Parm                    @InLib
 *====================================================================

 c                     Exsr       Init

 c                     Exsr       Exit

 *====================================================================
 * 1-off initialization
 *--------------------------------------------------------------------
```

continued on page 36

utility (which Part 2 will cover), we place the API call in the subroutine INIT at F (the reason for this will become clearer when we look at Figure 2). So, in the INIT subroutine, we

- reset the error data structure for the API call
- set the length to that of the data structure that will hold the returned data; we can specify a shorter length than the structure, provided the length covers the fields in the structure we want to use (if we specify a longer length, the API may "overflow" the actual length of the data structure and corrupt the storage area where the program stores its variables)
- tell the API which data format we'd like returned (in this case, we're choosing the first format, 'OBJD0100')
- specify the object name and library as a 20-byte field; this is common in APIs, and the data must be spaced out as 10 + 10 with no separators
- specify the object type

We then call the API and pass it our chosen data structure, the fields we've prepared, and the API error structure. After the call, we test the error structure to check for errors. If an error exists, we fail the program by calling *PSSR.

If we were to examine a dump that was produced by calling the program with an invalid library (or a library name that's not in all uppercase characters), we'd see that the data structure we input to the API — QUSD0100 — is unchanged (it has remained blank). The API has set the bytes available to 26, but the bytes provided remain at 16 — the difference being 10 bytes, which will hold the substitution data for the error message CPF9810 (it holds the name of the invalid library we requested) if the data structure is long enough.

FIGURE 1 *continued*

```
(E)
c     *Inzsr      Begsr
c                 Clear                   Qusec
c                 Eval      Qusbavl = 0
c                 Eval      Qusbprv = %len(Qusec)
c                 Endsr
*----------------------------------------------------------------
* initialization
*----------------------------------------------------------------
(F)
c     Init        Begsr
c                 Reset                   Qusec
c                 Eval      @Length = 90
c                 Eval      @Format = 'OBJD0100'
c                 Eval      @FilLib = @InLib  + 'QSYS'
c                 Eval      @ObjTyp = '*LIB'
c                 Call      'QUSROBJD'
c                 Parm                    Qusd0100
c                 Parm                    @Length
c                 Parm                    @Format
c                 Parm                    @FilLib
c                 Parm                    @ObjTyp
c                 Parm                    Qusec
c                 If        Qusbavl <> 0
c                 Exsr      *Pssr
c                 Endif
c                 Endsr
*----------------------------------------------------------------
* wrap up and go
*----------------------------------------------------------------
c     Exit        Begsr
c                 Eval      *inlr = *on
c                 Return
c                 Endsr
*----------------------------------------------------------------
* error routine
*----------------------------------------------------------------
c     *Pssr       Begsr
c                 Dump
c                 Endsr     '*CANCL'
*----------------------------------------------------------------
```

Calling APIs with Prototypes

Now, let's compare the example in Figure 1 with that in Figure 2, which employs prototyped calls. The prototype RtvObjD (at A) replaces the PLIST. You'll also see that I've defined the error data structure QUSEC in the program (at B) to replace the /COPY entry. This lets us set the initialization with the field definitions rather than in the C-specs, allowing us to remove the *INZSR subroutine. Also note that I've included another field (QUSMSGDATA) in this data structure for additional message data.

The call to the API has now been replaced with a more meaningful CALLP for RtvObjD (at C), which also uses expressions in the call. I think you'll agree that using prototyped calls makes the program clearer to read. Therefore, we'll use prototype calls when we build the utility in Part 2.

If you call this example with an invalid library name and examine the dump, you'll notice that within the QUSEC structure, the additional field QUSMSG DATA contains the library name. This is the substitution data for the CPF9810 message ID held in QUSEI.

Error Handling in APIs

Most of the APIs we're likely to use daily employ an optional parameter for controlling errors (e.g., as defined in the copy source member in Figure 1 at A). If we omit this parameter and the API detects an error, it crashes with an escape message. This means that if your

FIGURE 2
Calling a program with prototype calls

```
H debug
 *====================================================================
 * prototype for retrieving an object description
 *--------------------------
(A)
d RtvObjD          pr                  extpgm('QUSROBJD')
d ReturnArea                     1024    Options(*varsize)
d RtnLen                          10i 0 const
d Format                           8    const
d ObjFilLib                       20    const
d ObjType                         10    const
d ErrRtn                         256
 *====================================================================
 * API error data structure
(B)
d Qusec            DS
d Qusbprv                         10i 0 Inz(%size(Qusec))
d Qusbavl                         10i 0 Inz(0)
d Qusei                            7
d Quserved                         1
d QusMsgData                     240
 *====================================================================
 * API copy sources
 *--------------------------------------------------------------------
 * API structure for retrieving object description API
/copy QSYSINC/QRPGLESRC,QUSROBJD
 *====================================================================
 * API parameter fields
d @InLib           s                 10
 *====================================================================
 * Definitions
 *--------------------------------------------------------------------
c      *Entry      Plist
c                  Parm                    @InLib
 *====================================================================

c                  Exsr      Init

c                  Exsr      Exit

 *====================================================================
 * initialization
 *--------------------------------------------------------------------
(C)
c      Init        Begsr

c                  Reset                   qusec
c                  Callp     RtvObjD(Qusd0100 : %len(Qusd0100) :
c                             'OBJD0100' : @InLib + 'QSYS' : '*LIB' :
c                             Qusec)

c                  If        Qusbavl <> 0
c                  Exsr      *Pssr
c                  Endif

c                  Endsr
 *====================================================================
 * wrap up and go
 *--------------------------------------------------------------------
c      Exit        Begsr

c                  Eval      *inlr = *on
c                  Return

c                  Endsr
 *====================================================================
 * error routine
 *--------------------------------------------------------------------
c      *Pssr       Begsr

c                  Dump

c                  Endsr     '*CANCL'
 *--------------------------------------------------------------------
```

program doesn't handle the error, it will crash, too. It's far better to pass the parameter and look at the returned results so we can decide what to do.

Figure 1 at A is a copy member because the information is held as a data structure. As Figure 2 clearly shows, a data structure is constructed as follows:

- QUSEC is the name of the data

WORKING WITH APIs

structure.

- QUSBPRV indicates the bytes provided. It tells the API how long the data structure is. You typically specify a length of at least 16 bytes (the field can have values of 0, 8, or more). The API also updates this field with the number of bytes that can be returned if the data structure is long enough. (This is a four-byte binary field, not a four-digit field, so if you define it in RPG with digits, you need to specify a value between five and nine.)
- QUSBAVL shows the bytes available. Here, the API tells us how many bytes of error information is returned. The important point here is that if this value is zero, no error exists. If it's greater than zero, the API tells us how much information has been returned (again, this is a four-byte binary field).
- QUSEI specifies the exception ID (message ID). This is the message ID for the type of error that caused the API to fail (this is a seven-character field).
- QUSERVED is a single character reserved field, which we don't use.
- You can append to the data structure a field (which you name) for additional information that the API may return (specifically, the substitution data for the message ID). In Figure 2, I've named this field QUSMSGDATA. Whether or not you use it depends on your application. For comparison purposes, Figure 1 omits this field, but Figure 2 includes it.

Ready for the Next Step

Now that you've seen how calling APIs using PLIST and prototypes differs, you're ready for the next step — building the utility. In Part 2, we'll examine how to employ a user space, fill it with data, extract records, use other APIs to retrieve data, and more. So stay tuned! ■

Paul Morris *is a freelance senior analyst/programmer in the U.K. who provides programming and systems support for the iSeries. You can e-mail him at Paul@wssltd.demon.co.uk.*