Dynamic File Allocation From COBOL Programs

By Steve Comstock

Traditionally, when a COBOL program needs to access a file (sequential, VSAM, or, now, HFS (Hierarchial File System—in z/OS UNIX)), the pieces that have to be in place, at a minimum, are these:

1. Each file needs a SELECT statement that provides an internal name (called a "file name") and an external name (called a "DD-name") (these are placed in the Environment division—see FIGURE 1)
2. Each file needs an FD ("File Definition") entry in the File section of the Data Division.
   Each FD needs to be followed by a description of what records in the file look like. This can be a simple 01 item with just a length, or it can be a complex record layout. (see FIGURE 2)
3. In the procedure division, you must issue I/O verbs for your files, from this list:
   OPEN—connect your program file names to external files (more shortly)
   READ—retrieve a record from an external file into memory for processing
   WRITE—put out a record from memory to an external file
   REWRITE—update an existing record on a disk file (won't work for tape or other devices)
   DELETE—remove a record from a file on disk (won't work for tape or other devices)
   CLOSE—disconnect a program file name from an external file (this also flushes buffers, writes file labels, and other tasks)

Now when such a program is run in batch, it needs to have some JCL, including a JCL DD statement for every file that is opened by the program. There are a lot of options, but the general syntax is:

```
//ddname DD DISP=data_set_disposition,DSN=data_set_name
```

FIGURE 1: SELECT STATEMENT

```
Id division.
Program-id. Inq3223.

Environment division.
Input-output section.
File-control.
    Select trans-in  assign to queries.
    Select master assign to master.
    Select listing assign to listing.
```

Notes:

a. File names can be any non-reserved COBOL word (max of 30 characters)
b. DD-names need to follow z/OS rules for JCL DD statement names: 1-8 alphanumeric characters, first of which is alphabetic
c. You can use the same name for both the file name and DD-name, as long as the file name follows the more restrictive rules for DD-names
d. There are other possible clauses, but they are not relevant to this paper.

where "ddname" must match the DD-name used in the COBOL program. (Again, you must provide one such DD statement for each file to be opened.)

Historically, if you forgot to supply a DD statement (or if you misspelled the ddname), the OPEN statement would fail and the program would abend.

Later, a change was made to COBOL so that if you did not supply a DD statement with the correct name, COBOL would dynamically allocate the file for you!

Well, that sounds cool, but it leads to all kinds of problems. The main problem is that COBOL dynamically allocates the file as (NEW,DELETE), so when the program ends, the file is deleted! So here you are with a step that did not abend and you can't find your file.

More recent compilers have provided a run-time option so you can revert back to the old way—you can specify an LE parameter of CBLQDA(OFF) on your EXEC statement, for example:

```
//STEP22 EXEC PGM=INQ3223,PARM='/CBLQDA(OFF)'
```

Originally, the default was supplied as ON; more recently, the IBM-supplied default is OFF, so this whole conundrum may be moot for you.

On the other hand, sometimes you don't know the name of the data set you will be processing in advance, so you can't specify it on your JCL DD statement. For example, the name may be passed to your program by a calling routine, or in the PARM field of the JCL EXEC statement; or perhaps you have a routine that dynamically generates a name to ensure it is unique each time you run the job.

In the past, you had to forgo this capability, or call a subroutine written in Assembler. But the current COBOL compiler provides the ability to dynamically set the data set name to use for a file at run time. To make this work, you need to ensure two steps are taken:

1. Do *not* code a DD statement for the file you want to dynamically allocate in your run time JCL
2. In your program, before you open the file, you must set the data set name (and other parameters) into an *environment variable* that has the same name as the DD-name coded in your SELECT statement!

Well, the first part is easy enough. But what the heck is an environment variable? And how do you set it?

The concept of environment variables comes from the world of UNIX but is now pretty widely available in the classic z/OS world, too. An environment variable is a named memory location, managed outside your program, whose name is known to all the programs in the run-unit (that is, the variable's name is known to your program as well as to any subroutines, the LE runtime support routines, and so on; any of these programs can retrieve the value, or change it).

You create an environment variable in one of these ways:

- define it in your run-time JCL as another LE PARM variable, for example:

  ```
  //STEP22 EXEC PGM=INQ3223,
  // PARM='/ENVAR="QUERIES=MQCIC22.TEST.QS"'
  ```

- call the C function 'putenv', passing a null-terminated string consisting of an environment variable name (your choice, but it is case sensitive)

Since all currently supported compilers can call C functions directly, this is how we can accomplish dynamic file allocation from a COBOL program: set up an environment variable in our code, then call 'putenv'.

Although you can create any environment variable, in our case we will want to create an environment variable with the DD-name from our Select statement (and it has to be in upper case).

There are many possibilities, but we work with a simple example here. What we want to do is to create an environment variable named QUERIES with a value like this:

QUERIES=DSN(*dsn*) SHR

where *dsn* represents the name we want to obtain or create dynamically.

FIGURE 2: DESCRIPTION OF THE RECORDS IN THE FILE

```
Data division.
File section.
FD  trans-in
    recording mode is f.
01  trans-rec.
    02  trans-type        pic  xx.
    02  trans-part-no     pic  x(9).
    .
    .
    .

FD  master
    recording mode is f
    organization is indexed
    record key is master-part-no
    access is dynamic.
01  master-rec.
    02 master-part-no    pic  x(9).
    02 master-description pic x(30).
    .
    .
    .

FD  listing
    recording mode is f.
01  print-line       pic x(133).
```

Notes:
a. The name after FD must match the file name from the select statement.
b. Every file with a select needs an FD, and every FD must correspond to some select statement.
c. There are other possible clauses, but they are not relevant to this paper.

Suppose our program is a subroutine that is passed a parm containing the name of the data set we are to use, with at least one trailing space. That is, we know we will need the following Linkage Section:

```
Linkage section.
01 in-name pic x(55).
```

and our Procedure division header will need to look like this:

```
Procedure division using in-name.
```

(The length of 55 allows for the longest z/OS data set name (44 characters) plus possibly a member name (8 characters) bounded by parentheses (2 characters) followed by a one character space; not all the space is necessarily used, but this is the extreme case.)

Now we also need to define a place to build our string of "QUERIES=DSN(*dsn*) SHR", including room for the dsn we get from our input parm. Something like this will fill the bill:

```
01 file-name.
02 pic x(12) value 'QUERIES=DSN('.
02 dsname pic x(55) value spaces.
02 pic x(06) value z' SHR '.
```

Notice the last item's value clause is a null terminated string (the z before the quote tells COBOL to append a x'00' at the end). We do this because the 'putenv' C function requires a null-terminated string.

Now, we need to get our input parameter into the data item we named 'dsname', followed by a closing parenthesis. The STRING verb seems to be the right way to go, so:

```
string in-name delimited by space
')' delimited by size
into file-name
```

This gets the job done. Now, file-name will contain

```
QUERIES=DSN(dsn_from_parm) SHR x'00'
```

note that there may be many blanks between the closing parenthesis and SHR, but that is not a problem.

We're almost ready to call 'putenv'. But there is a problem: 'putenv' needs to have the address of this string passed. Since we are only passing one argument, COBOL wants to pass the address, but it also wants to turn on the end-of-list indicator (that is, it wants to set the leftmost bit in the address to 1; since this is a 31-bit address, it does not hurt to do this). But C routines do not use this convention, and they will not accept this address with the leftmost bit on. A conundrum.

To get around this, we will tell COBOL to pass a pointer, and that this pointer must be passed "by value." When you tell COBOL to pass a field by value, it just places the contents of the field in the parameter list and never turns on the end-of-list indicator. So, we need to define a pointer:

```
01 file-ptr pointer.
```

and we need to initialize it:

```
set file-ptr to address of file-name
```

and then we can call 'putenv':

```
call 'putenv' using by value file-ptr
```

But, there's another little thing: 'putenv' is a C function that returns a value. The returned value is 0 for success and -1 for failure. We probably want to know that. So we need to define a data item for holding this returned value. Maybe:

```
01 rc pic s9(9) binary value 0.
```

And, of course, we have to tell 'putenv' that this is where we want the returned value to be placed. Do this by using the 'returning' option of the CALL statement, so we have:

```
call 'putenv' using by value file-ptr returning rc
```

Then, perhaps we want to test that value after the call (or we can be supremely confident and assume it works).

Remember, now, that all this is done before OPEN. Now, when we open our file, we will be using the data set whose name was dynamically obtained!

The pieces are all assembled for you in FIGURE 3.

CONCLUSION

There are many other options you can specify for a data set that you determine dynamically. For example, you can create a new data set and

FIGURE 3: ASSEMBLED PIECES

```
Id division.
Program-id. Inq3223.

Environment division.
Input-output section.
File-control.
    Select trans-in  assign to queries.
    Select master assign to master.
    Select listing assign to listing.
.
.
.
Data division.
File section.
FD  trans-in
    recording mode is f.
01  trans-rec.
    02  trans-type        pic  xx.
    02  trans-part-no     pic  x(9).
    .
    .
    .

FD  master
    recording mode is f
    organization is indexed
    record key is master-part-no
    access is dynamic.
01  master-rec.
    02 master-part-no    pic  x(9).
    02 master-description pic x(30).
    .
    .
    .

FD  listing
    recording mode is f.
01  print-line       pic x(133).
.
.
.
Working-storage section.
01  file-name.
    02             pic x(12) value 'QUERIES=DSN('.
    02  dsname     pic x(55) value spaces.
    02             pic x(06) value z' SHR '.
01  file-ptr   pointer.
01  rc     pic s9(9)  binary  value 0.
.
.
.
Linkage section.
01  in-name    pic  x(55).
Procedure division using in-name.
.
.
.
    string in-name delimited by space
           ')'     delimited by size
           into file-name
    call 'putenv' using by value file-ptr returning rc
    if rc = -1 then
      display 'Error on putenv'
      display 'Program terminated'
      move 100 to return-code
      stop run
    end-if
    open input trans-in
.
.
.
```

specify SPACE parameters, or you can reference an HFS file, and do on. All the details and options are spelled out in the documents referenced below.

REFERENCES

The main sources for this article are available for reading or downloading for free from the Web at:

http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/IGY3LR20/CCONTENTS

http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/IGY3PG20/CCONTENTS

Steve Comstock is a self-employed trainer with Trainer's Friend Inc.