

Dynamic File Allocation with COBOL

With the advent of COBOL 2.2.0 and Language Environment (LE), the COBOL compiler has finally entered the 21st Century. It is now possible to perform dynamic file allocation within a COBOL batch program without calling an external SVC99 routine.

With the advent of COBOL 2.2.0 and Language Environment (LE), the COBOL compiler has finally entered the 21st Century. It is now possible to perform dynamic file allocation within a COBOL batch program without calling an external SVC99 routine.

When I heard that it was possible to do this within the confines of the COBOL system, I searched in vain for an `ALLOCATE` verb. Surely IBM had to have added such an extension to the language. However, that was not the case. To dynamically allocate a file in a COBOL batch program without JCL requires a strange marriage between LE's variables and the COBOL `ASSIGN` clause.

Unfortunately, you cannot simply refer to a single manual that succinctly puts all of these pieces together for you. It is a far more painful process that involves digging through a multitude of manuals and other resources to see if someone else has already conquered this beast. In my case, even the Web came up dry, so I turned to IBM for help. They provided me with all of the necessary information that eventually resulted in this article.

ENVIRONMENT VARIABLES

First, let me describe environment variables and the role they play in the dynamic allocation process. These variables can be defined and made available not only to the program that set them, but to any program that is subsequently called. If you have had any exposure to UNIX, you should be familiar

with this concept. The trick is to load an environment variable with the required file allocation information. This variable is then used as the external file reference that appears at the end of the COBOL `assign` clause.

At open time, the COBOL system first looks to see if this name refers to a `DD` statement in your JCL. If it cannot find it in the JCL, the COBOL system looks for an environment variable of the same name. If it cannot find the variable, the file status is set to 35, indicating that the required `DD` statement is missing, and so the open fails. If the variable is located, but the allocation information contained in it is incorrect (i.e., a syntax error or a misspelled dataset name), then the file status is set to 98 and again, the open fails. Finally, if the environment variable has been set up with a valid `allocate` statement, then the system will successfully `allocate` and `open` the file.

The file is automatically deallocated when it is closed, therefore, you are not required to repeat the aforementioned process with a `FREE` statement.

After the `close` is executed, you are free to update the environment variable with new file information and `open` yet another file via the same `ASSIGN` clause. Note that if you do this, all of the files that you `allocate` using this same variable must possess the same attributes (i.e., `LRECL`, `BLKSIZE`, etc.).

The allocation information that is placed in the variable closely resembles the information that you would normally code on a TSO `ALLOCATE` command. Details of this can be found in the *COBOL Language Reference* manual referenced

To dynamically allocate a file in a COBOL batch program without JCL requires a strange marriage between LE's variables and the COBOL ASSIGN clause.

at the end of this article. For now, take it for granted that the format required to allocate an existing file is DSN(file name) disposition. For example, the contents of an environment variable to allocate INPUT.PARMLIB would be DSN(INPUT.PARMLIB) SHR. It is worth noting at this point that you not only have the ability to allocate an existing file, but you can also create a new file by simply using the appropriate allocate parameters.

SETTING AN LE ENVIRONMENT VARIABLE

So, how do you set an LE environment variable? One way is through the PARM parameter on the EXEC statement within your JCL. For example, if you want to create an environment variable called DYNFILE that contains the information necessary to allocate INPUT.PARMLIB, you would code it as follows:

```
//STEP1 EXEC PGM=TESTPGM,PARM=('/ENVAR
("DYNFILE=DSN(INPUT.PARMLIB) SHR"))'
```

The forward slash (/) separates the run-time options from the program parameters. Note that for COBOL, the run-time options appear to the right of the slash. This is the opposite of the format defined by LE. You can change this order via the CBLOPTS installation default.

The ENVAR run-time option is used to establish an environment variable. In this case, I chose the name DYNFILE, but it can be anything you like within the restrictions that I will outline later. In the previous example, DYNFILE is being loaded with the necessary information required to allocate a dataset called INPUT.PARMLIB. While this is a perfectly legitimate method to set a variable, it really has little value if your intent is to dynamically allocate a file. After all, if you have to modify the JCL to identify the file you are going to allocate, why not just add the appropriate DD statement?

Dynamic allocation derives its power from a program's ability to determine the file it wants to use at runtime. To that end, COBOL must issue a call to a C function known as PUTENV to create an environment variable containing the appropriate ALLOCATE command. As I mentioned previously, the name of this variable must be used as the external file reference that is placed on the end of the COBOL ASSIGN clause.

FIGURE 1: SAMPLE PROGRAM TO PERFORM DYNAMIC FILE ALLOCATION USING ENVIRONMENT VARIABLES

```
CBL NODYNAM,LIB,OBJECT,RENT,APOST
ID DIVISION.
PROGRAM-ID. DYNALLC.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN UT-S-DYNFILE
        FILE STATUS IS INPUT-FILE-STATUS.

    SELECT PRINTER ASSIGN TO UT-S-SYSPRT.
*****
DATA DIVISION.
FILE SECTION.
*
FD INPUT-FILE
RECORDING MODE F
BLOCK CONTAINS 0 RECORDS
LABEL RECORDS ARE STANDARD.
01 INPUT-RECORD                                PIC X(80).

FD PRINTER
RECORDING MODE F
LABEL RECORDS ARE OMITTED
RECORD CONTAINS 133 CHARACTERS.
01 OUTREC.
02 OUT-LINE                                    PIC X(133).

*****
* WORKING STORAGE SECTION *
*****
WORKING-STORAGE SECTION.
01 INPUT-FILE-STATUS                            PIC 9(2).
01 RC                                            PIC 9(9) BINARY.
01 ADDRESS-POINTER                            POINTER.
01 FILE-ENVIRONMENT-VARIABLE                  PIC X(31)
    VALUE 'DYNFILE=DSN(INPUT.PARMLIB) SHR'.

PROCEDURE DIVISION.
*****
* MAIN ROUTINE *
*****
    SET ADDRESS-POINTER TO ADDRESS OF
        FILE-ENVIRONMENT-VARIABLE.
    CALL "PUTENV" USING BY VALUE ADDRESS-POINTER
        RETURNING RC.
    IF RC NOT = 0 THEN
        DISPLAY 'PUTENV FAILED'
        STOP RUN
    END-IF
    OPEN OUTPUT PRINTER.
    OPEN INPUT INPUT-FILE.
    READ INPUT-FILE.
    MOVE INPUT-RECORD TO OUT-LINE.
    WRITE OUTREC.
    CLOSE INPUT-FILE.
    CLOSE PRINTER.

GOBACK.
```

This limits your options with regard to giving your environment variable a name. It must adhere to the same syntax rules that are applied to the assignment name on a COBOL ASSIGN clause. That is, the name must contain one to eight upper-case

alphanumeric characters and the leading character must be alphabetic. Beyond that, you can name it anything you like.

The best way to put all of these pieces together is by showing you the sample program shown in Figure 1.

The first order of business is to set up the proper working storage items. Of particular note are ADDRESS-POINTER and FILE-ENVIRONMENT-VARIABLE. I chose these names for clarity, but you are free to use any identifiers you like.

ADDRESS-POINTER is defined as a pointer type and is used in the procedure division to contain the address of our environment variable, aptly named FILE-ENVIRONMENT-VARIABLE. As defined in the *COBOL Language Reference*, a pointer data item is a 4-byte elementary item. In other words, it is a 4-byte address and it cannot have any subordinate items.

The FILE-ENVIRONMENT-VARIABLE field contains the now familiar text used to dynamically allocate a file. The allocation will not take place until the program issues an open for INPUT-FILE.

In the procedure division, a SET verb is used to establish the addressing required for the subsequent call statement. At the completion of the SET, the 4-byte address of FILE-ENVIRONMENT-VARIABLE is contained in ADDRESS-POINTER. You are now ready to call PUTENV.

The format of the CALL statement is important, so let us examine it in more detail. When calling a C program from COBOL, you must adhere to specific linkage conventions. Therefore, the CALL statement uses "BY VALUE" to establish how variables are passed and "RETURNING" to provide a location where the return code can be placed.

BY VALUE is used primarily to call non-COBOL programs such as C, although it is perfectly legal to use it in a call to a COBOL routine. When you pass a parameter by value, you are sending a temporary copy of the parameter to the called program. That is, the called program can modify the data contained in the variable, however, the changes are not reflected in the calling program. Compare this to using the "BY REFERENCE" clause where changes to the parameter in the called program affect the original parameter in the caller.

The "RETURNING" phrase is, obviously, setting up a return code to be placed in the RC variable. This only works when the called program has a matching RETURNING associated with the entry point being called. For example, in COBOL it would look like the following:

```
PROCEDURE DIVISION USING BY VALUE vari-  
able-1 RETURNING variable-2.
```

The PUTENV program has been coded with the equivalent C syntax.

Finally, the CALL must be static. In other words, you must compile your program with the NODYNAM compile-time option. That means that you must have the CEE.SCEELKED library in the SYSLIB concatenation in the linkedit step. This is almost certainly a default in your shop.

Although IBM made it clear to me that NODYNAM was a requirement, I decided to give the DYNAM option a try anyway. After all, IBM could be wrong. The result

was a very ugly SOC1 abend. So much for doubting IBM.


That is all there is to it. Remember that the minimum level of COBOL you will require is COBOL 2.2.0. Additionally, you will need PTF UQ47307 for LE 2.8 or PTF UQ47311 for LE 2.10.

ACKNOWLEDGEMENTS/ RESOURCES

My thanks to John (last name unknown) of IBM whose responses to my IBMLink questions were the basis for this article.

For more information on the new functionality of the COBOL ASSIGN clause, refer to the *COBOL Language Reference* (SC26-9046-04). Note the revision level of this manual is 04. It must be at least this or higher.

For more information on environment variables, refer to the *Language Environment Programming Reference* (SC26-3312).

The PUTENV function is described in the *C/C++ Run-Time Library Reference* (SC28-1663). Note that a solid knowledge of C programming is required to get the full value from this manual. 

NaSPA member Ed Watson is a systems programmer with UtiliCorp United, Inc. in Omaha, NE. In the past 20 years, Ed has worked on VSE, VM, OS/390, AIX and HP/UX systems.