VSE Data Spaces: Part III
Using the ALESERV Macro

BY LEO J. LANGEVIN

In the first two parts of this three-part series, I examined what data spaces are, how the control blocks fit together, and demonstrated how to define a data space. In this concluding column I will demonstrate how to access a data space that has already been defined by use of a special value called a space token, or "STOKEN."

As we saw, the DSPSERV macro will allow a data space to be defined, released, and even extended. This is all fine and good, but how do you use this data space? What do you need to get data in or out of that space? We need to be able to point to that area and use it, so what's the answer to this formula? The key is the ALESERV macro and access registers. If you don't know anything about access registers, don't worry. I'll be covering them in great detail in the next few issues. For now, just think of an access register as another register that you can use, in conjunction with a general program register, to point to a different address space. For example, let's say that you are using general register 3, and access register use is active, and you load access register 3 with a special address space token, or STOKEN. Whatever data that you are accessing that is pointed to by register 3 may be going into a completely different address space, or partition. This includes reading or writing data. (Of course, I am only using register 3 in this statement as an example!) The ALESERV macro makes using access registers in conjunction with data spaces that much easier.

The DSPSERV macro will allow a data space to be defined, released, and even extended. This is all fine and good, but how do you use this data space? What do you need to get data in or out of that space?

Access registers are required in order to access address or data spaces. They are loaded with an access list entry table address, or ALET. By getting the ALET and loading the access register with that value, we can read and modify the storage used by the data space.

There are six different options available to the programmer when using the ALESERV macro: ADD, ADDPASN, DELETE, EXTRACT, EXTRACTH, and SEARCH. ADDPASN and EXTRACTH are not meant to be used by the normal application program and were created for IBM systems use.

Normally, you will only use the ADD option to get the ALET for a specific STOKEN. You might also use DELETE to delete the ALET; however, be careful to only delete the ALET for a data space that your program has defined. Deleting an ALET for a data space that you do not own,although documented as inoperable, can be done but will produce some exciting results! (This is especially true if you delete the ALET for the virtual disk that contains the label area. Trust me on this one, folks!)

You can also SEARCH a specific access list to see if an ALET is already defined, EXTRACT an ALET that is already defined, and ADDPASN to define an ALET for a PASN. EXTRACTH is an interesting option because while the other five options are documented in the ALESERV macro, this one is not. I believe that there is some special system use for this option.

In most normal circumstances you will probably only use ADD and possibly DELETE. Remember, your only need for ALESERV is to get the ALET associated with a specific STOKEN. When you start using some of the more esoteric functions of ALESERV, you are going outside of the normal data space access design, which is beyond the scope of this month's topic.

ADDING AN ALET

In most cases you will only need to code the ALESERV macro with the following options to ADD an entry:

```
[NAME] ALESERV ADD,
                    STOKEN=stoken,
                    AL=WORKUNIT,
       ALET=alet
```

The STOKEN will always be needed for every ALESERV ADD, EXTRACT, and SEARCH option. It will point to the special identifier for the specific data space.

AL is either WORKUNIT or PASN. WORKUNIT will always be used for data space access. When using this macro for address space definitions, PASN would be used.

ALET is an output field that is defined as the address where the ALET will be returned after ALESERV processing is complete.

One option that is in the ALESERV macro is the "RELATED="

keyword. This does nothing but provide some additional documentation for the author.

DELETING AN ALET

In most cases you will only need to define the ALESERV macro with the following options to DELETE an entry:

```
[NAME]  ALESERV DELETE,ALET=ALET
```

All you need to do is provide the ALET and the system will do the rest.

EXTRACTING AN ALET

Let's say that you have the ALET, but you don't have the STOKEN. What do you do? You would use the EXTRACT option to tell the system to insert the STOKEN value for you. Following is the format for this request:

```
[NAME] ALESERV EXTRACT,
         STOKEN=stoken,
         ALET=alet
```

ALET is your input field, while STOKEN is your output field.

SEARCHING FOR AN ALET

You can also search for an existing ALET instead of defining a new one, which is especially critical when you are doing a lot of data space access. However, given the current number of partitions available, this should not be a concern.

The general method of finding an ALET would be to code the following:

```
[NAME] ALESERV SEARCH,
          STOKEN=stoken,
          AL=WORKUNIT,
          ALET=alet
```

The system will use STOKEN as an input field and ALET as an output field, returning the value of the desired ALET.

OTHER OPERANDS

As a side note, there is another operand to the ALESERV macro that exists but is rarely used. It is the "ACCESS=PUBLIC" or "ACCESS=PRIVATE" selection. The default is to allow public access to that ALET. For security reasons, you may decide to use ACCESS=PRIVATE.

As with the DSPSERV macro, I did not discuss the use of the "MF" operand. During the rewriting of ALESERV, the "MF" function did not translate well to normal assembler and therefore the normal default is being used.

As you can see, the ALESERV macro utilization is dependent upon the existence of the data space. This is why ALESERV is performed after a DSPSERV request and why ALESERV is performed before a data space delete request. So let's use this information and take it to the next logical step.

USING DATA SPACES IN YOUR APPLICATION

Now that you know how data spaces are defined and deleted, and how ALETs are defined and deleted, let's build an application that can make use of this information. But first, here's an important point: Normal data spaces will only exist for the duration of the job that defined it. This means that if you define a data space and your job terminates, the data space is automatically deallocated. There are ways around this, however. The two simplest methods are to either have the program defining the data space run most of the time (such as a CICS application), or use the virtual disks provided by IBM as your personal data spaces. If you do this, just remember to make sure that there isn't any JCL that will use that virtual disk, thinking that it is a disk drive. If this happens, then the batch program will abend.

We need to be able to perform sequential bi-directional "I/O" as well as random access within the data space. This is why we have READS and WRITES for forward sequential access, READR and WRITER for reverse sequential access, and READ and WRITE for reading and writing to a specific spot in the data space.

Another important thing to remember about VSE data spaces is that they will disappear when you IPL your VSE operating system. Therefore, you do not want to keep important, modified, but yet-to-be stored data in a data space unless it is concurrently being written on a disk or tape file as well.

When you define a data space, you will need to know if you want other partitions to access it. You will also need to know if you want to EXTEND the data space. Additionally, you will need to use SCOPE=SINGLE if you want to protect your data from being accessed elsewhere or if you want the ability to extend the data space at a later time.

In the DSPACE subroutine, which is available for download from the NaSCOM Internet server as filename MAY98001.ZIP, I have not provided EXTEND processing because we default to

CA-FLEE and IBM-LIBR Bug Update

Author's Note: In my February 1998 column (which I wrote in December 1997), I said that there was a known and documented bug with CA-FLEE and IBM-LIBR. This has since been corrected and is no longer a problem. The reported problem has since been pulled off of IBM's list of known bugs as of late January 1998. A representative from CA reported to me that they have been getting a lot of calls concerning this, so I feel that this updated information needs to be provided to the VSE community.

using the definition of SCOPE=COMMON. If you want to allow DSPACE to perform EXTEND processing, you will need to modify the ADD default and include an initial block size as well as a maximum block size in the definition. Because I wanted to provide a simple solution to a common need, I did not do this. However, since you have the source code ...

When you define a data space, you access the area by using two registers: the access register and the general register. The access register points to the data space while the general register points to the area within that data space. This means that if you want to define 80-byte records and read the third record stored in that data space, your general register will need to contain a value of 160. To look at the first record, it will need to contain a value of zero. The DSPACE subroutine works with record sizes and record numbers. This eliminates the need to perform these types of calculations.

We need to be able to perform sequential bi-directional "I/O" as well as random access within the data space. This is why we have READS and WRITES for forward sequential access, READR and WRITER for reverse sequential access, and READ and WRITE for reading and writing to a specific spot in the data space.

In order to make this process easier to understand, I have provided DSPTEST. This file can be downloaded from the NaSCOM Internet server as filename MAY98001.ZIP and performs the following:

1. creates a data space
2. performs "QUERY DSPACE,ALL"
3. tries to create a replicate DSPACE, which should fail
4. loads the data space with 1,000 "random" records
5. reads the data that was created in step 4
6. reads the data in reverse after EOF was reached in step 5
7. deletes the data space
8. tries to delete the data space again, which should fail
9. performs "QUERY DSPACE,ALL"
10. ends test ts

Leo Langevin is the lead developer for NFS for VSE from Connectivity Systems. He has been involved with VSE since its inception. He can be reached at LEO@TCPIP4VSE.COM.