Introduction to Program Segmentation [4]

When using the loader, you specify the segment structure and the contents of the segments to be loaded. This section describes the principles of the loader program segmentation. The information in this section does not apply to nonsegmented programs. "Examples," page 123, contains an example of a segmented program.

In addition to automatic segment loading and unloading, the loader lets you do the following:

- Modify the segmentation structure, usually without recompilation.
- Overlay different modules (subroutines) without making significant source code changes.
- Define the contents of a segment by specifying only one module per segment.
- Pass arguments between subprograms residing in different segments.
- Unload segments and any contained data blocks. The loader then reloads the blocks with their updated images.

SEGLDR segment tree concept

4.1

The loader arranges program segments in a tree structure, as shown in Figure 1, page 56. A nonsegmented program consists of only the root segment.

Each segment in a tree contains one or more subprogram modules, and possibly some common blocks. Subprogram hierarchy helps you determine the shape of your tree.

The *root* segment of a tree is the predecessor for every branch segment and has no predecessor segment itself. Predecessor and successor segments lie on a common branch. Down the tree (or branch) means moving away from the root segment, and up the tree or branch means moving toward the root segment.

During program execution, only one immediate successor segment of each segment can be in memory at one time. The root segment is always memory-resident; other segments occupy higher memory addresses when required. Predecessor segments of the executing segment are guaranteed to be memory-resident. In addition, successor segments might be memory-resident, depending on recent subroutine calls to successor segments.

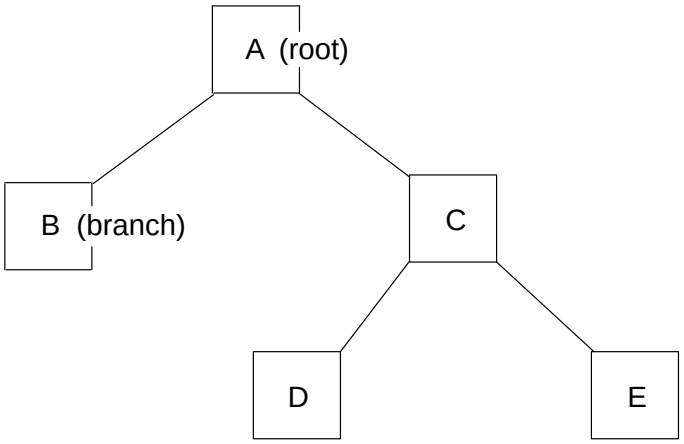

Figure 1. Segment tree

Each segment in Figure 1 is assigned an arbitrary but unique 1- to 8-character segment name.

The apex of the loader segment tree (segment A in Figure 1) is the root segment. The remaining segments (B, C, D, and E) are the branch segments.

Within these branch segments, B, C, D, and E are successor segments of A. B and C are immediate successor segments of A, and D and E are immediate successor segments of C. It follows, then, that C and A are predecessor segments for D and E, and A alone is the predecessor segment for B and C. C is the immediate predecessor segment of D and E.

Loader segment tree design
4.2

The only restriction on the height or width of the segment tree is that no more than 1000 segments, including the root, can be defined. A valid segment tree, however, must adhere to the following rules:

- Each segment tree can have only one root segment (a segment with no predecessor segments) and must have at least one branch segment.
- Each nonroot segment can have only one immediate predecessor segment.

Figure 2 and Figure 3 show valid segment trees.

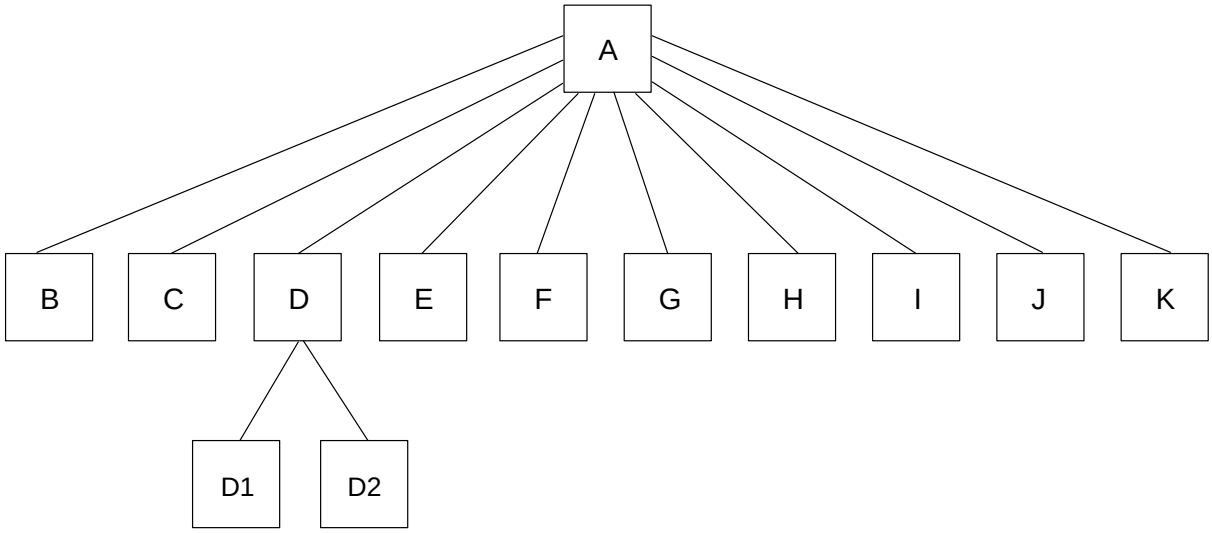

Figure 2. Valid segment tree (broad)

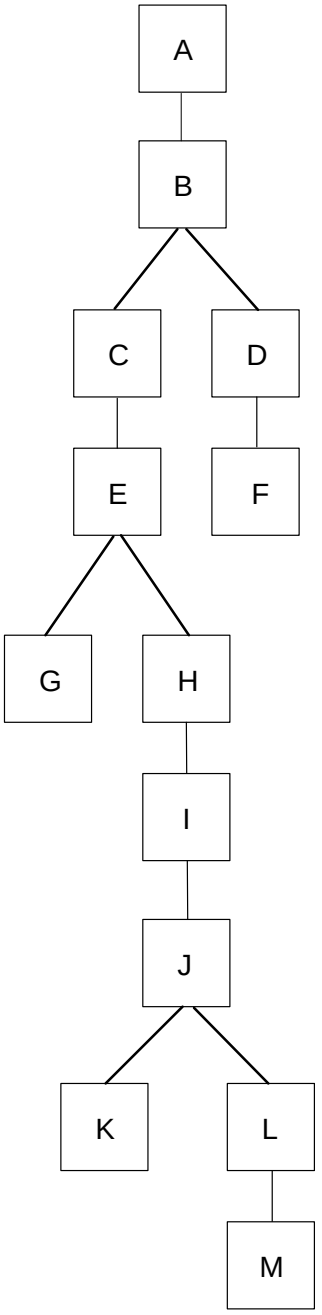

Figure 3. Valid segment tree (deep)

Figure 4 and Figure 5 show tree structures that are invalid because of their multiple root segments or multiple immediate-predecessor segments.

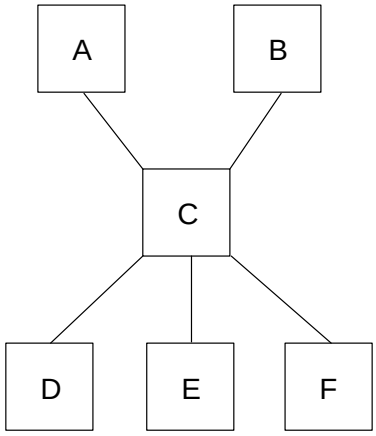

Figure 4. Invalid segment tree (multiple root segments)

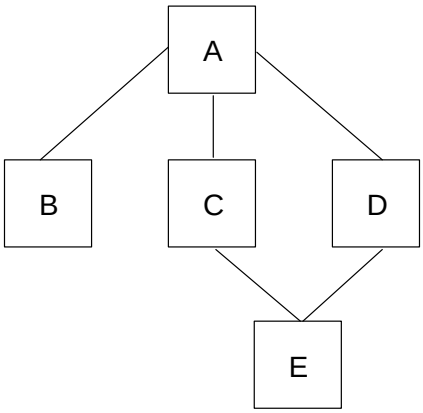

Figure 5. Invalid segment tree (multiple immediate-predecessor segments)

Subroutine calling between segments

4.3

Calls can be made from any module in a segment to any module (subroutine or function) in a successor or predecessor segment. Calls across the segment tree are invalid (see Figure 6, page 61). That is, subroutine calls can be made both up and down the tree if the calling and called modules are owned by segments on a common branch. If a call is made to a subroutine from a segment that is not an immediate predecessor to the segment

containing the subroutine, all intermediate segments on the branch are read into memory. In Figure 3, for example, if a line of code in segment I makes a call to a subroutine in segment M, segments J, L, and M are all read into memory.

When a call is made from a subroutine to a subroutine further down the branch at execution time, the loader does the following:

1. Intercepts the call
2. Loads the appropriate segment or segments (if not already in memory)
3. Jumps to the called entry point

The loader intercepts only calls to subroutines in successor segments because they are the only calls that can cause a segment to be loaded (if a segment is in memory, all of its predecessors (callers) are already in memory).

Caution: In CAL, it is strongly recommended that you use the `CALL` and `CALLV` macros for subroutine calls to other modules. If you do not do this, the calls between segments may fail, with unpredictable results.

Do not pass an entry point to a subroutine as an argument if the entry point is not in the same segment or a predecessor segment. In Fortran, for example, the following two statements can produce calls to segments not in memory:

```
EXTERNAL SUB1
CALL SUB (SUB1)
```

The segment `SUB1` may not be in memory when this call is made because the loader cannot detect runtime references.

You should not use the segment structure shown in Figure 6, because it generates an execution error (explanation following).

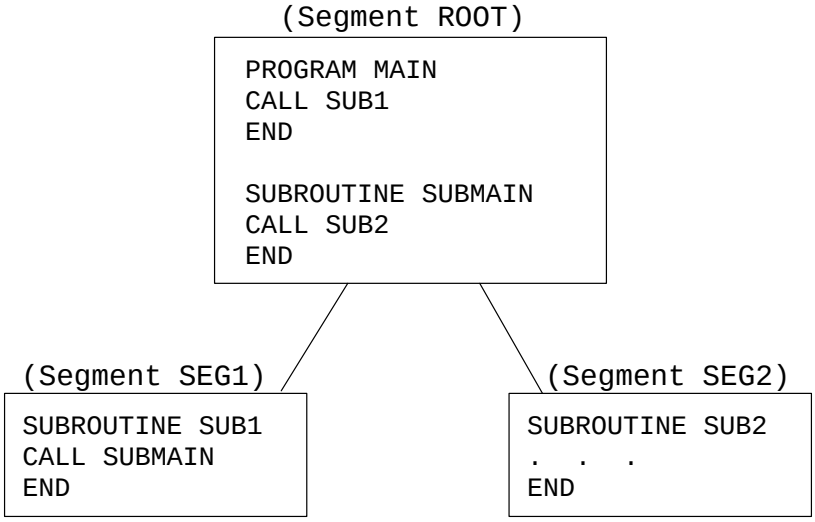

Figure 6. Invalid segment tree (call across segment tree)

Figure 6 shows an invalid segment structure that results in the following sequence of actions:

1. When SUB1 is called, segment SEG1 is read into memory.
2. When SUB2 is called, segment SEG2 is read into memory, overwriting SEG1.
3. On the return to SUB1 from SUBMAIN, SEG1 is no longer in memory; therefore, control cannot return to SUB1.
4. $SEGRES terminates the program at this point, displaying an error message.

The loader handles subroutine calls as shown in Figure 7.

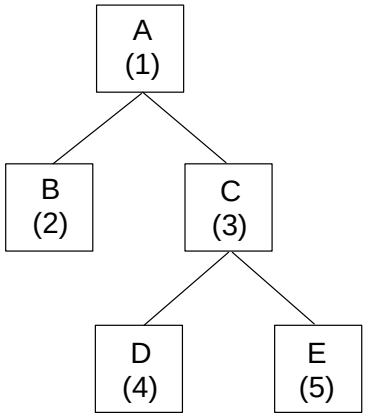

Figure 7. Valid and invalid subroutine references

The following subroutine call descriptions are related to the tree structure shown in Figure 7. Numbers 1 through 5 represent modules in segments A through E.

From	To	Head
1(A)	2,3,4,5	Valid; may need to load some segments.
2	1	Valid; no load needed.
2	3,4,5	Invalid; calls across a branch.
3	2	Invalid; calls across a branch.
3	1,4,5	Valid; may need to load a segment if the call is to module 4 or 5.
4	5,2	Invalid; calls across a branch.
4	1,3	Valid; no load needed.
5	4,2	Invalid; calls across a branch.
5	3,1	Valid; no load needed.

Using segmentation with multitasked programs
4.4

You must be careful when combining segmentation and multitasking in the same executable program, because there is a significant risk of program failure. If a program is multitasked, it is possible for one task to call a subroutine that initiates a segment change, while another task is actively executing in that segment. To avoid this situation, it is necessary to restrict multitasking activity to areas of the program in which segment changes will not occur.

Macrotasking involves partitioning large areas of a program into tasks, so that the tasks can run on several CPUs simultaneously. Because the program tasks contain many subroutines, it is more likely that a segment change will be initiated somewhere within a tasked region of the program. The use of macrotasking in a segmented program is strongly discouraged.

Usually, the tasking activity for an autotasked program is contained within a particular subroutine, although references to other routines are possible. Segment changes are unlikely to occur within tasked regions of the program. If references to other routines are made, you should ensure that all routines within the multitasked region are contained within a single segment.

For more information on multitasking, see the *CF77 Optimization Guide*, publication SG–3773.