Segmentation Directives [5]

This section describes the directives you need for defining the memory tree structure of your program and for assigning modules and common blocks to specific segments. All of the directives in “Segment tree definition directives” and “Segment description directives,” page 66, are segment directives, and they must be placed after all global directives. “Examples,” page 123, contains an example of a segmented program.

Segment tree definition directives
5.1

Use the TREE and ENDTREE segment tree definition directives to tell the loader the shape of the tree that represents the memory layout of your code. Tree structures can be of any width or depth, but they must contain no more than 1000 segments. Only one set of TREE and ENDTREE directives is allowed in a program load.

The TREE directive signals the end of the group of global directives (described in “General Directives,” page 21) and the beginning of the segment tree definition directives. The set of directives specifying the tree structure follows TREE.

The ENDTREE directive terminates the segment tree definition directives; it signals the end of the tree description. The ordering of segment tree definition directives between TREE and ENDTREE is unimportant. The segment description directives immediately follow ENDTREE.

Tree definition directives apply only to segmented programs.

Format:

```
TREE
segname(segname1[,segname2,segname3,...,segnamen])
ENDTREE
```

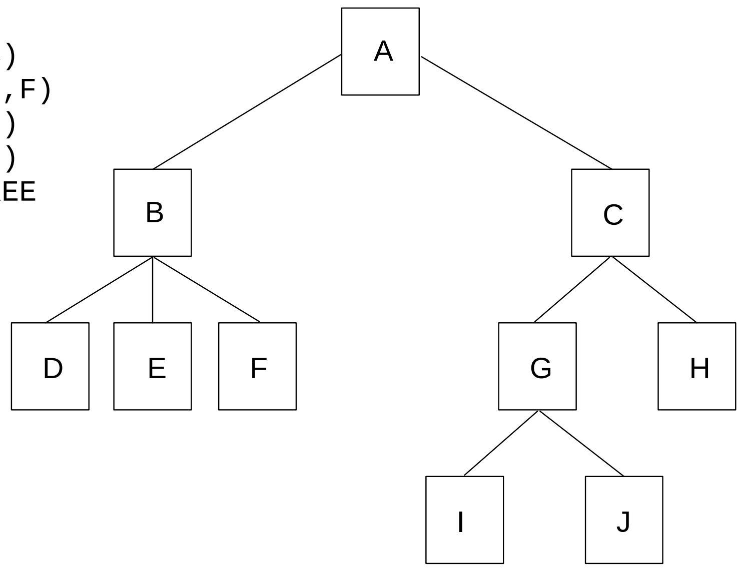
segname Name of a segment.

$segname_i$ Names of all immediate successor segments of *segname*.

If the description of a segment continues beyond one line, end each continued line with a comma.

Example:

```
TREE
A(B,C)
B(D,E,F)
C(G,H)
G(I,J)
ENDTREE
```



Segment description directives
5.2

Segment description directives apply only to segmented programs and specify the contents of the segments. At least one module or common block must be assigned to each segment.

In addition to the directives described in this subsection, the `COMMENT`, `ECHO`, and `TITLE` directives discussed in “General Directives,” page 21, can also be used within the segment description directives.

`SEGMENT` *and* `ENDSEG` *directives*
5.2.1

The `SEGMENT` directive specifies the segment being described by the segment description directives. `SEGMENT` is always the first of the segment description directives, except when you are using the `DUP` directive.

The `ENDSEG` directive terminates the segment description. Any of the segment description directives may appear between `SEGMENT` and `ENDSEG` in any order.

Format:

```
SEGMENT=segname
seg descr dirs
ENDSEG
```

segname	1- to 8-character segment name.
seg descr dirs	One or more segment description directives.

Example (the `//` indicates blank common):

```
SEGMENT=SAM
  MODULES=A,B,C
  COMMONS=//,SAMCOM
ENDSEG
```

`MODULES` *and* `SMODULES` *directives*
5.2.2

The `MODULES` and `SMODULES` directives let you assign modules to the segment specified by the `SEGMENT` directive. The `MODULES` and `SMODULES` directives also order the modules within the segment.

You must assign at least one module to each segment, and you may assign as many as needed. You do not need to assign all modules to segments. “Program Duplication and Block Assignment,” page 75, describes the way the loader handles modules that you have not explicitly assigned to segments. Modules that should be assigned explicitly include those that should reside in the segment specified by the `SEGMENT` directive but are called by modules in predecessor segments.

If you use the `MODULES` directive, an error message is issued if the modules specified cannot be located in any included file. Error messages are not issued if `SMODULES` is used.

Format:

```
MODULES=modname1[,modname2,...,modnamen]
```

modname$_i$ Names of the modules to be loaded.

You may specify argument *modname*$_i$ as either *modname* or *modname*:*name*. Use the second form to specify a module to be loaded from a specific file.

If your list of modules is greater than one line, you may use more MODULES directives or end the line with a comma and continue the list on the next line.

Example:

```
MODULES=SUBA,SUBB:lib1.a,SUBC
MODULES=SUBD:FILE.o
```

The loader obtains modules SUBA and SUBC from the first file in which each is encountered. It obtains SUBB from file lib1.a and SUBD from file file.o.

COMMONS *and* SCOMMONS *directives*
5.2.3

The COMMONS and SCOMMONS directives specify common blocks to be loaded into the segment specified by the SEGMENT directive. Common block specification is optional unless common blocks are to be duplicated or loaded in a specific order.

Common blocks with the same name that are loaded into two or more segments are considered unique. They occupy different memory locations, and the program can reference their contents unambiguously.

You may not include the dynamic common block in a COMMONS directive, because it is not assigned to a segment. See “Common block use,” page 83, for more information on common blocks.

If you use the COMMONS directive, an error message is issued if the indicated common blocks cannot be located in any included file. No error messages are issued if SCOMMONS is used.

Format:

```
COMMONS=blkname1[:size1][,blkname2[:size2 ],...,blkname_n[:size_n]]
```

blkname$_i$ Name of the common blocks to be loaded.

size$_i$ Decimal number indicating the size of the common block. If present, it overrides any common block sizes declared in your code. If the size specified is 0, the first common block size encountered in your code (for this common block) is used. By default, the loader uses the longest common block definition it encounters in your code as the size of the common block.

Common blocks are loaded in the order in which they are specified. The effect of multiple `COMMONS` or `SCOMMONS` directives is cumulative.

If you continue this directive beyond one line, end each continued line with a comma.

`BIN` *directive*
5.2.4

The `BIN` directive specifies files containing relocatable modules. The loader loads all modules within the specified `bin` files into the segment specified by the `SEGMENT` directive.

Format:

```
BIN=bin1[,bin2,bin3,...,bin_n]
```

bin$_i$ Names of files containing relocatable object modules.

The loader processes the files in the order presented. The effect of multiple `BIN` directives is cumulative.

If you continue this directive beyond one line, end each continued line with a comma.

Example:

```
SEGMENT=SEG1
BIN=seg1a.o,seg1b.o
BIN=seg1c.o
seg1d.o,seg1e.o
ENDSEG
```

In this example, all modules in files `seg1a.o`, `seg1b.o`, `seg1c.o`, `seg1d.o`, and `seg1e.o` are loaded into segment `SEG1`.

SAVE *directive*
5.2.5

The `SAVE` directive specifies whether the current segment state is written to mass storage before the loader overlays it with another segment. This directive overrides the effect of the global `SAVE` directive for individual segments.

> **Caution:** If you do not use the segmented `SAVE` directive and if you have not specified `SAVE=ON` as a global directive, `SAVE=OFF` is assumed. If the `SAVE` directive is `OFF` when a segment is loaded into the same memory area as the current segment, the updated values in the current segment are lost.

If you specify `SAVE=ON`, however, the loader writes the updated image of the overlaid segment to mass storage before the new segment is loaded. Subsequent execution of a saved segment starts from its saved image. This lets you overlay data areas whose updated values are required in subsequent executions of the saved segment.

Format:

```
SAVE=ON|OFF
```

`ON`	Enables segment saving.
`OFF`	Suppresses segment saving (default).

For an example of the use of this directive, see "SAVE directive," page 72.

DUP *directive*
5.2.6

Use the DUP directive if you want modules with the same name to be loaded into different segments. The DUP directive must precede all SEGMENT directives when duplicate module names are to be loaded.

You can duplicate the modules by using the DUP directive or by using the MODULES directive and assigning the same module name to more than one segment. "Program Duplication and Block Assignment," page 75, discusses the handling of duplicate modules and entry points in detail.

Format:

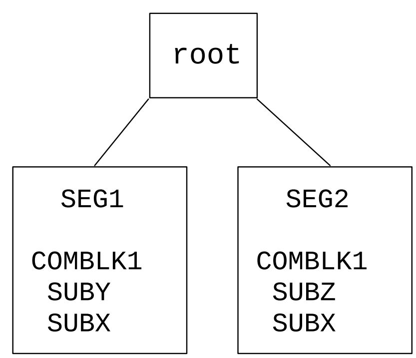
DUP=*modname*(seg_1[,seg_2,...,seg_n])

modname Name of a module to be loaded into more than one segment.

seg_i Names of the segments in which *modname* is to be loaded.

Example:

```
DUP=SUBX(SEG1,SEG2)
SEGMENT=SEG1
MODULES=SUBY
COMMONS=COMBLK1
ENDSEG
SEGMENT=SEG2
MODULES=SUBZ
COMMONS=COMBLK1
ENDSEG
```



In this example, assume that the module name and entry-point name are the same. Module SUBX is duplicated in segments SEG1 and SEG2. If SUBY is to call SUBX in segment SEG1, SUBY must be assigned to segment SEG1. If SUBZ is to call SUBX in segment SEG2, SUBZ must be assigned to segment SEG2. If SUBY or SUBZ were to go into root, the call would be ambiguous.

Global directives for segmentation
5.3

The directives in this subsection are global directives; that is, they must be specified before the TREE directive and they affect the entire program. These directives apply only to segmented loads.

SLT *directive*
5.3.1

The SLT directive specifies the size of the Segment Linkage table (SLT). The loader's resident run-time routine uses the SLT to service intersegment subroutine calls. The loader writes the actual SLT requirement to the listing file upon load completion. If SLT specifies a size less than the actual requirement, an error message specifies the actual requirement.

Format:

```
SLT=nnn
```

nnn Size (decimal word count) to be reserved for the SLT.

By default, the loader computes the size of the SLT according to the following formula: SLT=40*NBRNCH; NBRNCH is the number of nonterminal segments (segments having at least one successor segment). Calls to predecessor segments need no resident loader intervention.

SAVE *directive*
5.3.2

The global SAVE directive determines whether the current segment states are written to mass storage before they are overlaid with another segment. The global SAVE directive suppresses or enables saving of all segments, but the local SAVE directive can override the global SAVE directive for individual segments.

When SAVE=ON, the loader writes the updated image of the overlaid segment to mass storage before the new segment is loaded. Subsequent execution of a saved segment starts from its saved image; this lets you overlay data areas whose updated values you require in subsequent executions of the saved segment.

If the SAVE directive is OFF when a segment is loaded into the same memory area as the current segment, the updated values in the current segment are lost.

Format:

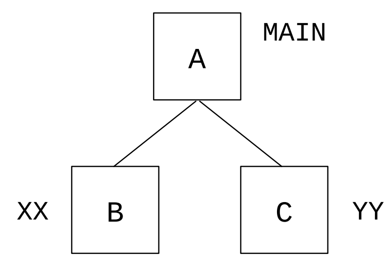
```
SAVE=ON|OFF
```

ON Enables segment saving.

OFF Suppresses segment saving (default).

Example:

```
SAVE=ON
TREE
A(B,C)
ENDTREE
SEGMENT=A
MODULES=MAIN
SEGMENT=B
MODULES=XX
SEGMENT=C
MODULES=YY
ENDSEG
```



The preceding example program performs calculations on two large data arrays, X(100000) and Y(100000), contained in subroutines XX and YY, respectively. It completes part of the calculations on one array, then on the other, then returns to the first, and so on, alternating between them. Because the arrays are in two separate subroutines that are never active at the same time, the two arrays can be overlaid rather than forced to the root segment (A).

COPY *directive*
5.3.3

The COPY directive forces your program to execute from a scratch file. This enables $SEGRES to use a faster form of I/O, which may speed program execution, but increase program start-up time. Programs in which the same segments are loaded and executed many times may improve their performance.

COPY has no effect if SAVE=ON for any segment, because SAVE also forces the use of a scratch file.

Format:

```
COPY=ON|OFF
```

ON	Program executes from scratch file, using a faster I/O method.
OFF	Disables execution from scratch file (default).

SEGORDER *directive*
5.3.4

The SEGORDER directive lets you determine the order of the segments in an executable file. Ordering the segments can speed up program execution, particularly when part of the file can be contained in buffer memory.

Format:

SEGORDER=seg_1,seg_2,...,seg_n

seg_i	Name of a program segment.

The loader writes the segments to the executable file in the order specified. The root segment is always first, regardless of the SEGORDER specification. You do not need to specify all program segments in the SEGORDER directive; segments not specified follow the specified segments in the order in which they are specified in the directives.