Central Memory Allocation by SEGLDR [9]

This section describes the different techniques that the loader uses to allocate user code and data into central memory on various Cray Research systems. Generally, you do not need to know about the techniques that the loader uses, because the default for your system is selected to work for most applications. For some applications, you may need to override the loader defaults, and this can be done using the directives described in "Program alignment and initialization," page 47.

If your application depends on any particular memory allocation scheme, it is recommended that you generalize the program to remove this dependency. Such code is nonstandard, and such dependencies can hinder maintenance of the code over time as systems change.

You can use the `ORDER` directive to specify the memory allocation scheme you desire. This works as long as you do not try to run your code on a different Cray Research system that does not support the specific option. Cray Research has changed and added memory allocation algorithms in the past, and will continue to do so, with the aim of improving the ease-of-use, system throughput, and performance of Cray Research systems. Applications that depend on specific memory allocation schemes will likely not be stable over time.

Definitions of terms
9.1

The following terms are used in this section:

Term	Definition
Block	The unit in which compilers and assemblers generate code and data for the loader to load. The actual memory size of a block is determined by the program.
Code block	A block containing nothing but instructions.
Common block	A block equivalent to the entity defined by a Fortran COMMON statement or C global data item.
Initialized ... block	A local data or common block that has initial values assigned by the program (as with the Fortran DATA statement).
Local data block	A block containing statically allocated local data.
Mixed block	A block containing both instructions and local data.
Uninitialized ... block	A local data or common block with no initial values assigned by the program.

Executable program organization
9.2

Every UNICOS executable program is organized in three sections: the text section, the data section, and the BSS section. Normally, the text section contains instructions, the data section contains initialized static data, and the BSS section contains uninitialized static data. Only the text and data sections are written into the executable file. The BSS section of the program is allocated at execution time. The various allocation methods

attempt to maximize placing uninitialized blocks into the BSS section whenever allowed by the hardware and the constraints of the allocation scheme. The placement of blocks into either the text or data section is critical only for shared text programs.

`ORDER` directive 9.3

`ORDER` is a global directive. It lets you control the central memory allocation method used by the loader.

Format:

```
        ⎡TEXT,DATA,BSS⎤
ORDER=  ⎢   SHARED    ⎥
        ⎣   SS.TDB    ⎦
```

The operation of each allocation scheme is described in the following paragraphs.

`SHARED` Separates the program code and data into two distinct address spaces and collates each one. `ORDER=SHARED` is used to create shared text programs that execute on Cray PVP systems under the UNICOS operating system, but `ORDER=SHARED` is not allowed on other systems. The program cannot contain any blocks of mixed code and data if this option is to be effective.

`TEXT,DATA,BSS`
Allocates code (`TEXT`) blocks, followed by initialized data (`DATA`) blocks, followed by uninitialized data (`BSS`) blocks. This is the default.

`SS.TDB` Creates a split-segment program and allocates code (`TEXT`) blocks, followed by initialized data (`DATA`) blocks, followed by uninitialized data (`BSS`) blocks. See “Memory allocation for segmented programs,” page 99, for more information.

Note: ORDER=SHARED cannot be used with segmented applications. ORDER=SS.TDB cannot be used with nonsegmented applications.

Command-line equivalents: –n and –O options

TEXT,DATA,BSS allocation scheme for memory allocation
9.4

The TEXT,DATA,BSS allocation scheme is the default on Cray PVP systems. The TEXT,DATA,BSS scheme allocates memory in the following order:

1. Code blocks
2. Initialized local data blocks
3. Initialized common blocks
4. Uninitialized local data blocks
5. Uninitialized common blocks

The TEXT,DATA,BSS scheme assigns as many uninitialized blocks as possible to the BSS section of the program.

Shared-text allocation scheme for memory allocation
9.5

The SHARED allocation scheme can be used to create shared text programs. In order to create a shared-text program, all object modules used in the program must be split into fully-separated code and data blocks. All Cray Research compilers generate separate code and data blocks and all Cray Research libraries contain separated modules. If you include your own assembly language routines, however, you must ensure that the generated code is separated from other modules in your program by including CODE and DATA attributes in any SECTION pseudo-instructions. If all modules are separated, the loader loads all the code sections of the program into one address space, and then loads the DATA and BSS sections into a separate address space.

Advantages of shared-text programs 9.5.1

A shared-text program has two major advantages:

- Multiple processes using the same application can share code, while keeping separate data areas. Thus, process demands on central memory are reduced.
- When the UNICOS operating system allocates memory for a process, it must find sufficient contiguous memory to allow the process to execute. With split code and data, the amount of memory required for the process is the same, except it is divided into two smaller pieces. Therefore, the UNICOS operating system can search for two small sections of memory rather than a single large one.

Disadvantages of shared-text programs 9.5.2

A shared-text program has two major drawbacks:

- The CDBX debugger cannot operate on shared-text programs. You should not use the shared-text scheme while debugging the program.
- Shared-text programs cannot be segmented.

Memory allocation for segmented programs 9.6

The allocation orders specified by the ORDER directive allocate each segment as a contiguous area of memory. Each segment is allocated separately; the modules and common blocks assigned to the segment are allocated in the specified order. Each segment begins where its predecessor ends.

On Cray PVP systems, code must reside in the first 4 Mwords of memory. Large data areas in the root segment may occupy enough memory below these limits to force code in later segments above these limits. To successfully load programs that encounter this problem, you can use the SS.TDB value for the ORDER directive.

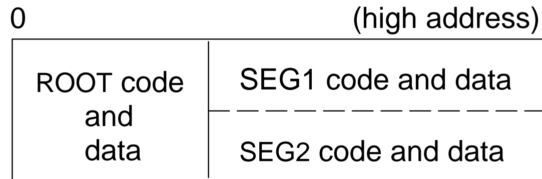
The SS.TDB allocation order creates split-segment programs. Each program segment is separated into a data section and a code section, which are allocated separately. Any modules and common blocks assigned to a segment are still allocated in the specified order. The code section of each segment is allocated in memory starting where the code section of the segment's predecessor ends. The data portion of each segment (except the

root segment) is allocated in memory following the data section of the segment's predecessor. The data section for the root segment is allocated after the highest address used to store code from the segments.

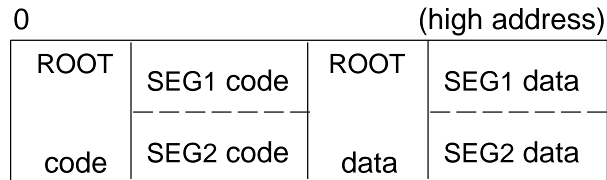
The following segment tree directives describe a program with a root segment and two successor segments:

```
TREE
ROOT(SEG1, SEG2)
ENDTREE
```

The MODULES,COMMONS, COMMONS,MODULES, and TEXT,DATA,BSS allocation orders create a program having the following structure in memory:



The SS.TDB allocation order creates a program with the following structure in memory:



ORDER=SS.TDB
9.6.1

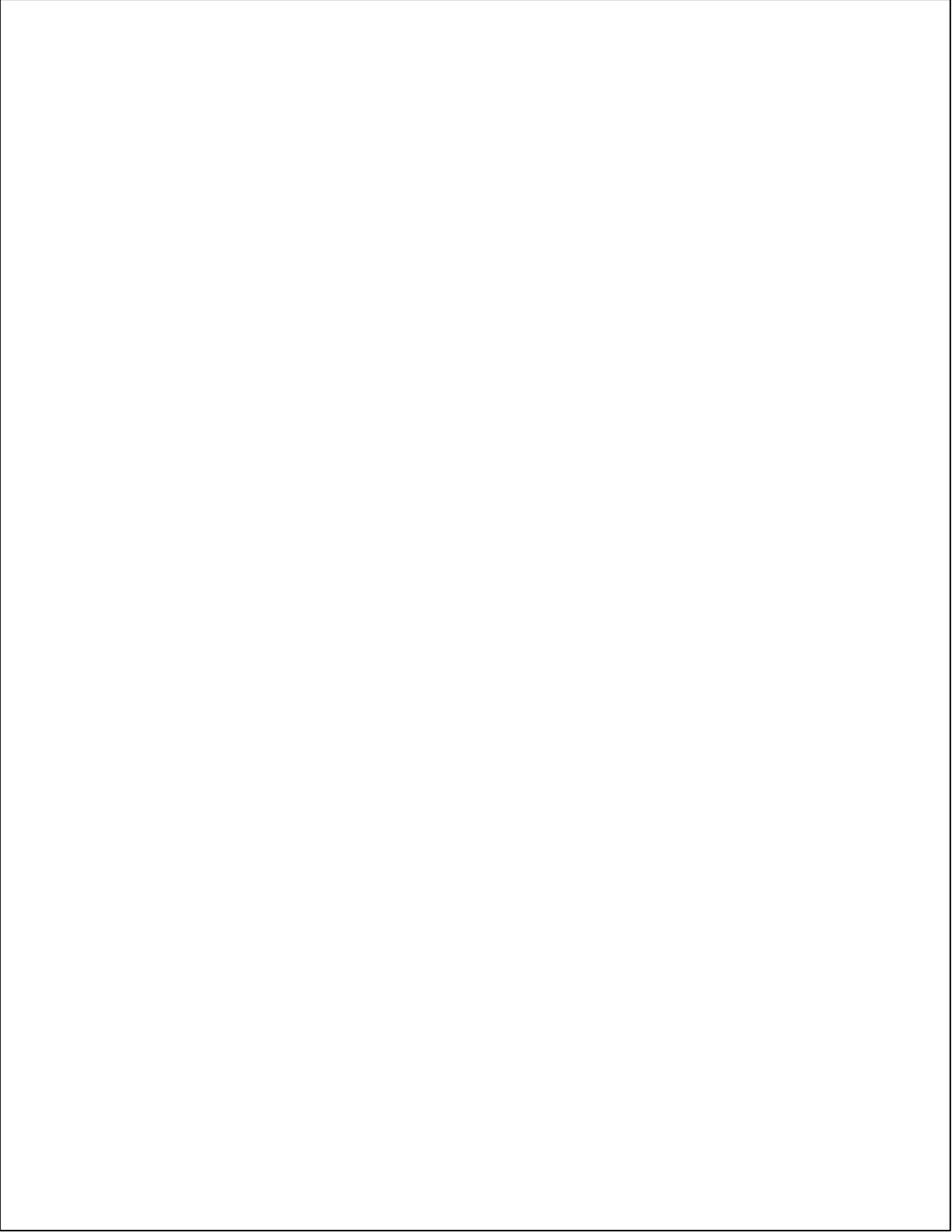
You should use the SS.TDB allocation order on Cray PVP systems when large data areas in the root segment of a program force code in successive segments above the 4-Mword memory boundary. The SS.TDB allocation scheme creates a split-segment program, allocating blocks to the code and data sections within each segment, as follows:

Code section:

- Code and mixed blocks

Data sections:

- Initialized local data blocks
- Initialized common blocks
- Uninitialized local data blocks
- Uninitialized common blocks



Soft Externals [10]

This section describes the special handling that the loader performs when processing "soft" references to an external symbol, or "soft externals."

Soft external references
10.1

Soft externals let the user control whether modules containing entry points to external functions or data objects are linked to the user's program. If the user program declares a reference to an external function as "soft," that reference is not sufficient to ensure that the external function will be included in the program. The function will be included only when referenced elsewhere in the program.

For example, Figure 11 contains two user programs, *flowpgm* and *noflwpgm*. *flowpgm* calls `flowtrace`, performs several functions, and then calls `exit`. *noflwpgm* does *not* call `flowtrace`, but it performs several functions, and then calls `exit`. The `exit` routine is called by both user programs; it processes `exit` calls for programs that call `flowtrace` and for programs that do *not* call `flowtrace`. Therefore, `exit` contains conditional calls to `flowexit`, which is an entry point within the `flowtrace` module. If `flowexit` is declared as a "hard," or normal external reference in `exit`, all of the `flowtrace` module must be loaded with each user program that calls `exit`, regardless of whether the user program calls `flowtrace`. If `flowexit` is declared as a soft external, then the `flowtrace` module is linked to the user program only when `flowtrace` is referenced. In Figure 11, the `flowtrace` module will be loaded with *flowpgm*, but it will not be loaded with *noflwpgm*.