This chapter describes the Segment Manager, the part of the Macintosh Operating System that loads and unloads your application's code segments into and out of memory. By dividing your application's executable code into segments, you allow it to run in a memory partition that is smaller than the total size of the application itself and the data it is using.

To use this chapter, you should already be familiar with the basic concepts of the Resource Manager and the Memory Manager. You need to know about the basic operation of the Resource Manager because segments are stored as resources. You need to know about the basic operation of the Memory Manager to understand when and why segments might be purged from memory. See the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory*.

You should read this chapter if your application contains multiple code segments that do not all need to be in memory at one time.

About the Segment Manager

Your application's executable code is stored in its resource fork as one or more resources of type `'CODE'`. These code resources are known as **segments** because the division of routines into code resources is controlled by segmentation directives you provide to your development system.

The Process Manager loads some code segments into memory when your application is launched. The Segment Manager loads other segments whenever you call any externally referenced routine contained in those segments. Both of these operations occur completely automatically and rely on information stored in your application's jump table and in the individual code segments themselves.

The Segment Manager loads segments into relocatable, purgeable blocks in your application heap. A segment is locked when it is first read into memory and at any time thereafter when routines in the segment are executing. This locking prevents the block from being moved during heap compaction and from being purged during heap purging.

Although needed code segments are loaded into memory automatically, it is your application's responsibility to unload any segments that are not currently being used. The Segment Manager provides a single procedure, `UnloadSeg`, that you can call to unload a segment. To **unload** a segment is simply to unlock it. By unlocking unneeded segments, you allow them to be relocated or purged if necessary to accommodate a later memory-allocation request. Thus, using the Segment Manager to unload unneeded segments is one important aspect of an efficient memory-management policy.

The following sections describe in detail the reasons for segmenting an application and the structure of the jump table.

Code Segmentation

Your development system's linker divides your application's executable code into segments according to directives that you provide. The **main segment** contains the main program. This segment is loaded into memory when your application starts to run and is never purged or unlocked as long as the application is running. The main event loop and other frequently needed small routines are generally stored in the main segment.

Most applications, however, consist of multiple code segments. There are two principal reasons for dividing code into different segments:

- **Compiler limitations.** Most development systems generate PC-relative instructions for intrasegment references (references to other routines within the same code segment). Because PC-relative instructions on an MC68000 use a 16-bit offset, the offset to the last routine in the segment cannot be larger than 32K bytes. Some development systems therefore restrict the size of any one code segment to 32K bytes.
- **Memory limitations.** Many applications are so large that the entire executable code, together with static data (such as your application's global data and resources) and data created dynamically during the execution of the application (such as windows and the items they contain), simply cannot fit into a memory partition of reasonable size.

By dividing your executable code into segments, you can circumvent both these limitations. The size of your application can increase as required to provide the desired capabilities without necessitating an increased run-time memory partition. For example, code that isn't executed very often (such as code for printing a document) can be put into a separate segment; it's loaded when needed and can be unloaded to free the memory for other uses when it's no longer needed.

Note
Some development systems allow you to create segments that are larger than 32K bytes. Consult your development system's documentation to determine how and when to increase segment size. ◆

The key fact to keep in mind when deciding how to group routines into segments is that an entire segment is loaded into memory whenever you call one of the routines in the segment. It makes sense, therefore, to group related routines in the same segment. You should segment routines according to your run-time call chain rather than on a simple file-by-file basis.

There are also some less obvious guidelines to follow when grouping routines into segments.

- Put your main event loop into the main segment.
- Put any routines that handle low-memory conditions into a locked segment (commonly the main segment). For example, if your application provides a grow-zone function, put that function into a locked segment.
- Put any routines that execute at interrupt time, including VBL tasks and Time Manager tasks, into a locked segment (commonly the main segment).

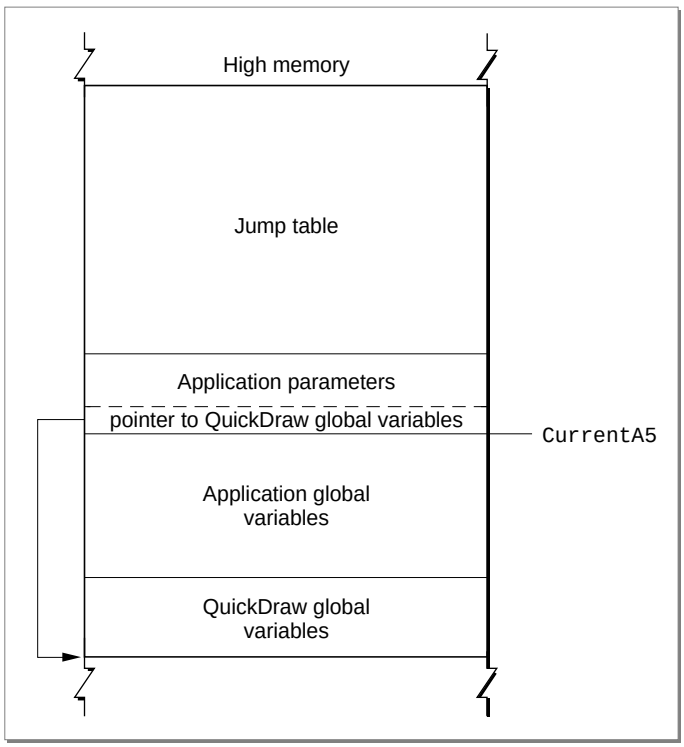
- Put into a separate segment any initialization routines that are executed exactly once at application startup time. Then unload that segment after those routines are executed. There is, however, at least one important exception to this rule. Routines that allocate nonrelocatable objects in your application heap should be called in the main segment, before you load any code segments that will later be unloaded. If you put such allocation routines into a code segment that is later unloaded and purged, you increase heap fragmentation. Routines such as `MoreMasters` and `InitWindows`, which are typically called at the beginning of an application, allocate nonrelocatable objects and should therefore be in the main segment.

The Jump Table

Note
This section describes how the Segment Manager works internally and is included for informational purposes only. You don't need this information to use the Segment Manager routine. Moreover, the information presented here might not be accurate for your development system. See the note on page 7-7. ◆

The loading and unloading of segments are implemented through your application's **jump table,** an area of memory in your application's partition that contains one entry for every externally referenced routine in every code segment of your application. The location of the jump table is illustrated in Figure 7-1.

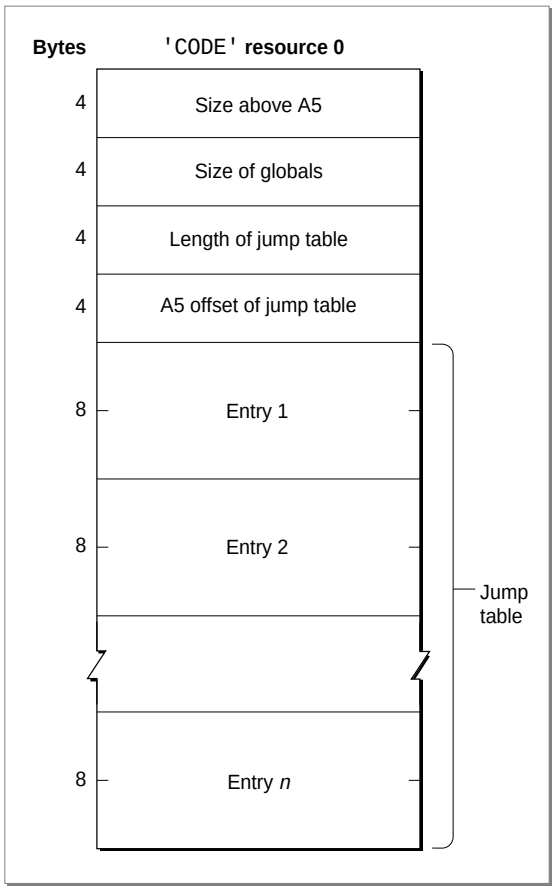
Figure 7-1 The location of the jump table



The jump table is accessed through the A5 register and is therefore part of your application's A5 world.

The jump table is created by your development system's linker and is stored in segment 0 of your application (which is the `'CODE'` resource with an ID of 0). Segment 0 is a special segment created by the linker for every application; it contains information about the A5 world and the jump table. Figure 7-2 illustrates the structure of segment 0.

Figure 7-2 The structure of segment 0



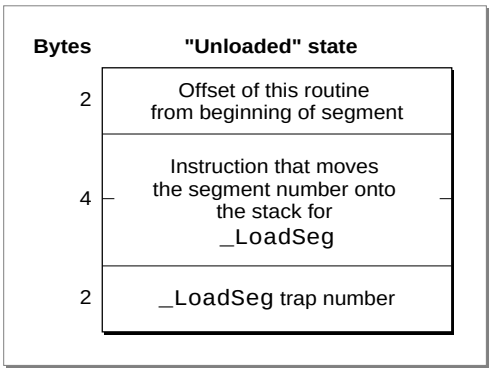
Segment 0 consists of these elements:

- Size above A5. The size (in bytes) from the location pointed to by register A5 to the upper end of the application space.
- Size of globals. The size (in bytes) of the application global variables plus the QuickDraw global variables.
- Length of jump table. The size (in bytes) of the jump table.

- A5 offset of jump table. The offset (in bytes) to the jump table from the location pointed to by register A5. This offset is stored in the global variable `CurJTOffset`.
- Jump table. A contiguous list of jump table entries.

When the MPW linker encounters a call to a routine in another code segment, it creates a **jump table entry** for that routine. (All entries for a particular segment are stored contiguously in the jump table.) The structure of a jump table entry varies according to whether the segment it references is loaded or unloaded. If the segment is not yet loaded into memory, the jump table entry has the structure illustrated in Figure 7-3.

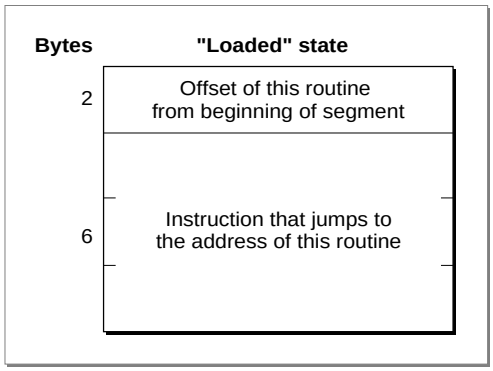
Figure 7-3 Format of an MPW jump table entry when the segment is unloaded



Note
Some development systems use a different format for jump table entries of unloaded routines to circumvent the 32K-byte limitation on the size of segments, global data, or the jump table itself. Consult the documentation for your development system to see whether it uses the jump table entry formats described in this section and whether you can safely call the `UnloadSeg` procedure (which changes jump table entries). ◆

The jump table refers to segments by segment numbers assigned by the linker. If the segment isn't loaded, the entry contains code that loads the segment. When a segment is unloaded, all its jump table entries are in the "unloaded" state. When a call to a routine in an unloaded segment is made, the code in the last 6 bytes of its jump table entry is executed. This code calls the `_LoadSeg` trap, which loads the segment into memory, transforms all of its jump table entries to a "loaded" state, and invokes the routine by executing the instruction in the last 6 bytes of its jump table entry. Figure 7-4 illustrates the format of a jump table entry in the "loaded" state.

Figure 7-4 Format of an MPW jump table entry when the segment is loaded



Subsequent calls to the routine also execute this instruction. When you call `UnloadSeg`, it restores the jump table entries to their "unloaded" state. Notice that the last 6 bytes of the jump table entry are always executed; the effect depends on the state of the entry at the time.

To set all the jump table entries for a segment to a particular state, the Segment Manager needs to know exactly where in the jump table all the entries are located. It gets this information from the **segment header,** 4 bytes at the beginning of the segment that contain the offset of the first routine's entry from the beginning of the jump table (2 bytes) and the number of entries for the segment (2 bytes).

Using the Segment Manager

The Segment Manager provides one routine for use by applications, the `UnloadSeg` procedure. You use this routine to unload code segments. The Operating System also provides two low-memory global variables that you can use to override the default segment-loading behavior and to monitor the system's automatic loading of code segments.

Unloading Code Segments

You can use the `UnloadSeg` procedure to unload segments. To unload a particular segment, pass `UnloadSeg` the address of any externally referenced routine contained in that segment. For example, to unload the segment that contains the procedure `DoPrintFile`, execute this line of code:

```
UnloadSeg(@DoPrintFile);
```

You can call `UnloadSeg` at any time except when you are executing code contained in the segment to be unloaded. A typical strategy is to unload all code segments except

segment 1 and any other essential code segments each time through your application's main event loop.

▲ **WARNING**
Before you unload a segment, make sure that your application no longer needs it. Never unload a segment that contains a completion routine or other interrupt task (such as a Time Manager task or VBL task) that might be executed after the segment is unloaded. Never unload a segment that contains routines in the current call chain. ▲

The `UnloadSeg` procedure does not actually remove the segment from memory. Instead, it unlocks the segment, thereby making the segment relocatable and purgeable. This permits the Memory Manager to relocate or purge the segment if necessary to gain some space in the application heap.

Loading Code Segments

The Segment Manager loads a code segment into memory automatically when you call any externally referenced routine in that segment. In most cases, the Segment Manager moves the block occupied by the code segment as high in the application heap as possible (by calling the Memory Manager procedure `MoveHHi`) and locks the block (by calling `HLock`) so that it cannot be moved or purged. You can disable or enable the call to `MoveHHi` and monitor the loading of segments into memory by manipulating two low-memory global variables.

If a code segment to be loaded is unlocked (that is, if it's not in memory and its `resLocked` attribute is clear, or if it is in memory and is unlocked), then the `_LoadSeg` trap calls the Memory Manager procedure `MoveHHi` to move the segment toward the top of the current heap. To prevent heap fragmentation, you should call the Memory Manager procedure `MaxApplZone` early in your application's execution. Otherwise, the heap will grow incrementally, and these automatic calls to `MoveHHi` may leave your code segments scattered throughout the heap. You can, however, disable the call to `MoveHHi` by setting the low-memory global variable `SegHiEnable` to 0. If this variable contains the value 0, `_LoadSeg` does not call `MoveHHi` to move the segment toward the top of the heap.

Occasionally, especially during application development, it is useful to monitor the otherwise largely invisible process of loading segments. You can do this by manipulating the system global variable `LoadTrap`. Before any routine in a newly loaded code segment is executed, the `_LoadSeg` trap inspects the `LoadTrap` global variable. If `LoadTrap` has a nonzero value, then `_LoadSeg` calls the `_Debugger` trap. This provides a useful way for you to monitor the loading of segments by the Segment Manager.

Segment Manager Reference

This section describes the routine provided by the Segment Manager.

Routine

The Segment Manager provides only one routine, the `UnloadSeg` procedure.

UnloadSeg

You can unload a segment by calling the `UnloadSeg` procedure.

```
PROCEDURE UnloadSeg (routineAddr: Ptr);
```

`routineAddr`
The address of any externally referenced routine in the segment to unload.

DESCRIPTION

The `UnloadSeg` procedure unloads a segment, making its storage relocatable and purgeable. You specify which segment to unload by passing the address of any externally referenced routine in that segment. The segment won't actually be purged until the memory it occupies is needed. If the segment is purged, the Segment Manager reloads it the next time one of the routines in it is called.

Note
The `UnloadSeg` procedure works only if called from outside the segment to be unloaded. ◆

Summary of the Segment Manager

Pascal Summary

Routine

```
PROCEDURE UnloadSeg          (routineAddr: Ptr);
```

C Summary

Routine

```
pascal void UnloadSeg        (void *routineAddr);
```

Assembly-Language Summary

Global Variables

`CurJTOffset`	word	Offset to jump table from location pointed to by A5.
`LoadTrap`	byte	If nonzero, call `_Debugger` before executing routine in a newly loaded segment.
`SegHiEnable`	byte	If nonzero, don't call `MoveHHi` when loading segments.

Advanced Routine

Trap macro	On entry
`_LoadSeg`	stack: segment number (word)