About Memory Management Overview of Memory Managment Principles

Several new features of memory management on Macintosh computers have been introduced in system 7.0, including changes to the **Memory Manager**-the part of the Operating System that controls the dynamic allocation of space in the application heap and the system heap. Two important additions to memory management in system 7.0 are support for 32-bit addressing and virtual memory. In addition, changes have been made under System 7.0 to the temporary memory routines became were already available under MultiFinder starting with system 6.0.

The information in this section supplements earlier sections on the **Memory Manager**, and it supersedes the discussion of temporary memory routines contained in Chapter 3 of the *Programmer's Guide to MultiFinder*.

The changes and additions to memory management on Macintosh computers described here are available only in system 7.0 or above. Support for 32-bit addressing is available only on machines with 32-bit clean ROMs (for example, the Macintosh IIci and the Macintosh IIfx). You can use the **Gestalt** function documented in **Compatibility Guidelines** to determine whether a machine was started up with 32-bit addressing and whether the enhanced temporary memory routines are available. Virtual memory is available only on machines equipped with a memory management unit (MMU). Currently, these machines include 68030-based machines (where the MMU is built into the CPU) as well as 68020-based machines that contain the 68851 Paged Memory Management Unit (PMMU). You can use the **Gestalt** function to determine whether virtual memory is installed. Most applications, however, do not need to know whether virtual memory is installed.

You need to read this and related topics if your application

- uses any of the temporary memory routines available under MultiFinder in system 6.0 and built into the Operating System in system 7.0

- has critical timing requirements, executes code at interrupt time, or performs debugging operations, any of which might be affected by the presence of virtual memory

- is not 32-bit clean (that is, does not operate correctly in an environment that uses the full 32 bits of a pointer or handle for memory addresses)

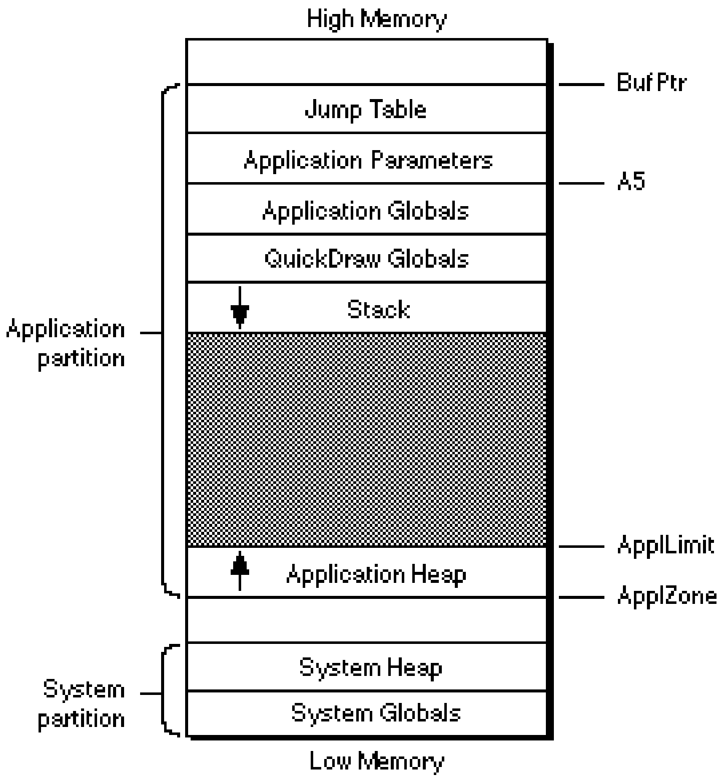
- uses customized window definitions (resources of type 'WDEF') or customized control definitions (resources of type 'CDEF')

- uses the **StripAddress** function documented in the **Operating System Utilities**

- installs routines (such as **Time Manager** tasks, VBL tasks, **Notification Manager** tasks, I/O completion routines, and so forth) that execute at interrupt time or at times when your application is not the active application

This section begins with an overview of memory management on Macintosh computers and explains the connections between cooperative multitasking, the **Memory Manager**, virtual memory, and your application's use of its own private memory. This overview also describes how the user controls various aspects of memory management through the Memory control panel.

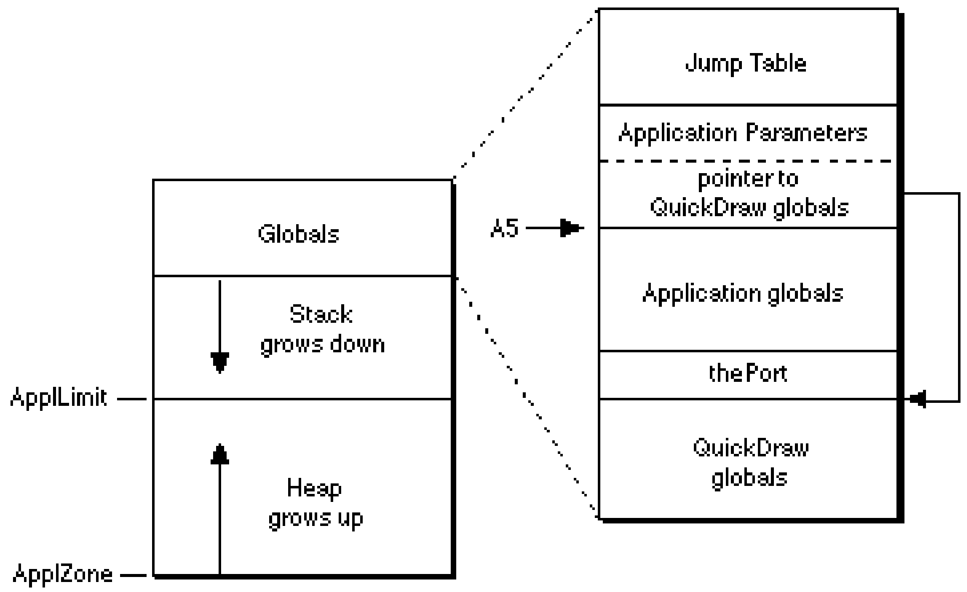
The Macintosh Operating System manages the loading of applications, desk accessories, resources, and other code into and out of memory. Prior to the introduction of MultiFinder in system 5.0, only one application could execute at a time. As a result, the organization of the available physical memory was relatively simple, as shown in the figure below. The available RAM was divided into two broad zones, a system zone and an application zone. The system zone, which resides at the bottom of memory, contains room for system global variables and a system heap. The system global variables maintain information about the operation of the Operating System itself (for example, the amount of time elapsed since the system was started up). The system heap contains the executable code for the Operating System, as well as some of the data structures used by the system software.



The memory organization in a single-application environment

When your application is launched, it is allocated a partition or heap of memory called the *application partition*. That partition must contain required segments of the application code as well as any other data associated with the application. You allocate space within your application's partition by making calls to the **Memory Manager**, either directly (for instance, using the **NewHandle** function) or indirectly (for instance, using a routine like **NewWindow** that calls **Memory Manager** routines).
The **Memory Manager** controls the dynamic allocation of space in your application's heap.

The application partition is divided into three main parts: an application heap (which holds the executable code of the application and perhaps some of the application's resources), a stack, and a set of parameters and global variables that are private to the application. This set includes the application's **QuickDraw** global variables, the application's own global variables and parameters, and a jump table. The divisions of the application partition are illustrated in the figure below.



The organization of the application partition

The Operating System keeps track of information in the application partition by storing various addresses in system global variables. For instance, the beginning of the application heap is stored in the global variable ApplZone. Similarly, the system global variable CurrentA5 points to the application parameters, the first long word of which is a pointer to the application's **QuickDraw** global variables.

Note that CurrentA5 points to a boundary between the application's parameters and its global variables, so the application's own global variables are found as negative offsets from the value of CurrentA5 This particular boundary is important because the Operating System uses it as a way of accessing your application's **QuickDraw** globals and application globals as well as the application parameters and the jump table, both of which are fixed distances from that boundary. This information is known collectively as the **A5 world** because the Operating System uses the microprocessor's A5 register to point to that boundary.

Note: An application's global variables may appear either above or below the **QuickDraw** global variables because the relative locations of these items are linker-dependent.