1

SYSTEM SOFTWARE, MEMORY, AND RESOURCES

Includes Demonstration Program SysMemRes

Macintosh System Software

All Macintosh applications make many calls, for many purposes, to Macintosh system software functions. Such purposes include, for example, the creation of standard user interface elements such as windows and menus, the drawing of text and graphics, and the coordination of the application's actions with other open applications.[1]

The majority of system software functions are components of either the **Macintosh Toolbox** or the **Macintosh Operating System**. In essence:

- Toolbox functions have to do with mediating your application with the user. They relate, in general, to the management of elements of the user interface.

- Operating System functions have to do with mediating your application with the Macintosh hardware, performing such basic low-level tasks as file input and output, memory management and process and device control.

Managers

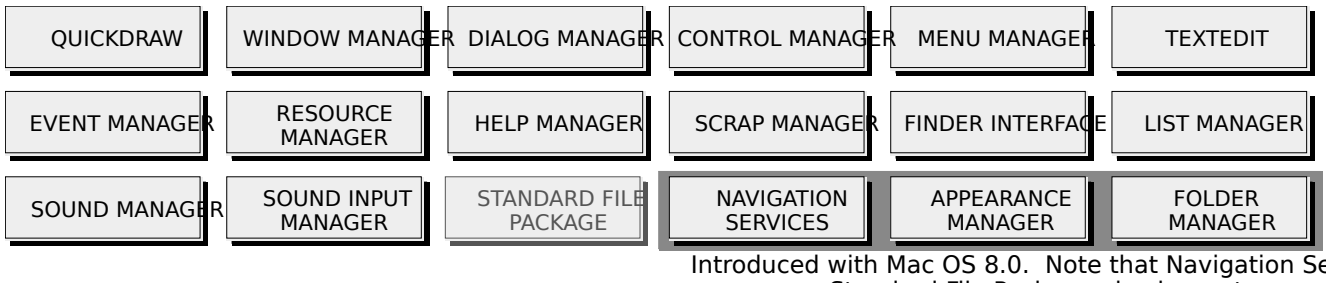
The entire collection of system software functions is further divided into functional groups which are usually known as **managers**.[2]

Toolbox

The main Toolbox managers are as follows:

1 The main *other* open application that an application needs to work with is the **Finder,** which is responsible for keeping track of files and managing the user's desktop. The Finder is not really part of the system software, although it is sometimes difficult to tell where the Finder ends and the system software begins.

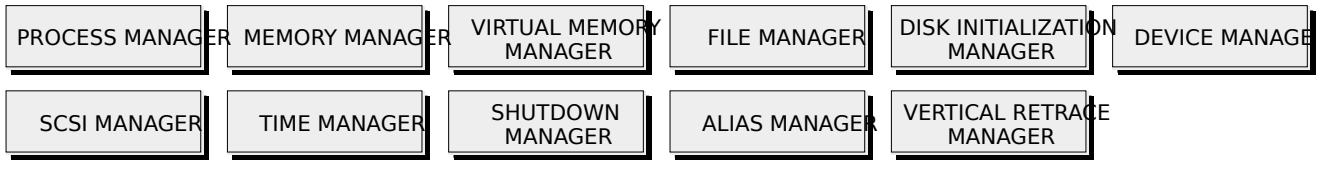
2 For historical reasons, some collections of system software functions are referred to as **packages.** In general, the distinction between managers and packages is unimportant. Packages are nowadays generally referred to as managers.



Introduced with Mac OS 8.0. Note that Navigation Se
Standard File Package obsolescent

Operating System

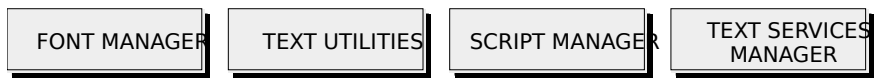
The main Operating System managers are as follows:



Additional System Software

The system software also includes a number of other components which do not historically belong to either the Macintosh Toolbox or Macintosh Operating System. These are categorised as follows:

- ***Text Handling.*** Text handling on the Macintosh is fundamentally graphic in that text is drawn as a sequence of graphic elements, and is designed to support more than one script (writing system). In addition to QuickDraw (see above), the components of the Macintosh script management system (that is, the essential text handling managers) include:



- ***Interapplication Communication.*** The interapplication communications architecture (IAC) provides a mechanism for communication between Macintosh applications. It includes:

- ***QuickTime.*** QuickTime is a collection of managers and other system software components which allow an application to control time-based data such as video clips, animation sequences and sound sequences. It includes:

- ***Communications Toolbox.*** The Communications Toolbox is a collection of system software managers which provide an application with basic networking and communications services. It includes:

System Software Functions

Functions in ROM

System software functions, which are also called **traps**, reside mainly in ROM (read-only memory). When an application calls a function, the Operating System intercepts the call and, ordinarily, executes the relevant code contained in ROM.

Functions in RAM — Patches

A **patch** is an additional component of the system software, and is usually stored in the System file, which is located in the System folder. To understand patches, some background is necessary.

The mechanism of the Operating System intercepting the call to the ROM-based code provides a simple way for the Operating System to substitute the code that is executed in response to a call to a particular trap.

All traps are numbered, and a **trap dispatch table** in RAM (random-access memory) matches each trap's number to its address. One advantage of this arrangement is that it makes it easy to correct bugs in the ROM-based code without replacing the ROM.[3] When a particular trap is called, the trap dispatch table can cause the Operating System to load some substitute code into RAM and execute that code instead of the ROM-based code. This RAM-based replacement code is called a patch**.**

Functions in RAM — Extensions

The System file also contains system software components which are *not* in ROM. These functions are like patches except that, when loaded, they do not replace existing ROM-based functions. The current method for adding capabilities to the system software is to include the code of the new functions as a system **extension.** Extensions are located in the Extensions folder and are loaded into memory at startup.

The Appearance Manager, a system software component introduced with Mac OS 8, is unique as a manager in that it is actually delivered as an extension.

Glue Functions

Some functions declared in a particular development system's header files are provided by the development system itself, not by the system software. These functions are known as **glue functions** and are constructed by modifying available system software functions in some way. Knowing whether a particular function is implemented as glue code is generally only relevant to low-level assembly level debugging.

Memory

In the Macintosh's cooperative multitasking environment, an application can use only part of the available RAM. When the Operating System starts up, it divides the available RAM into the **system partition** and the remainder of RAM, the latter being available for applications or other software components. When an application is launched, the Operating System allocates to it a section of RAM known as an **application partition**. Application partitions are loaded into the top part of RAM first.

Organisation of Memory - 680x0 Run-Time Environment

Macintosh computers use the Motorola 680x0 microprocessor. Power Macintosh computers use the PowerPC microprocessor. Although there are broad similarities, the organisation of memory in the 680x0 microprocessor run-time environment[4] differs from that in the PowerPC microprocessor run-time environment.

---

3 The other key advantage of this arrangement is that it overcomes the unavoidable difficulty of maintaining the same address for a particular trap as newer versions of the system software are developed. (It is easy to keep the trap number the same over time but difficult to ensure that its address remains forever unchanged.)

4 A run-time environment is a set of conventions which determine how code is to be loaded into memory, where it is to be stored, how it is to be addressed, and how functions call other functions and system software routines.

Fig 1 illustrates the arrangement of memory in the 680x0 run-time environment when several applications are open at once. Basically, the system partition occupies the lowest memory address space and the remaining space is allocated to the Process Manager, which creates a partition for each open application.

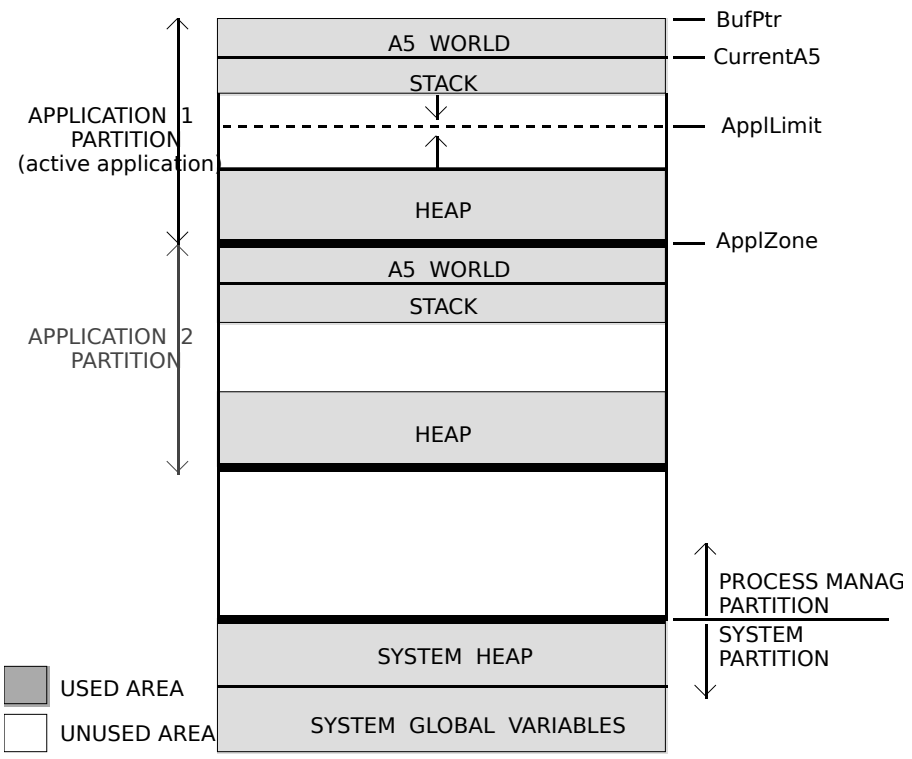


FIG 1 - MEMORY ORGANIZATION - 680x0 RUN-TIME EN

The System Partition

System Heap

The **system heap** (see Fig 1) is reserved for the exclusive use of the system, which loads into it various items such as system resources, system code segments, and system data structures. System patches and system extensions are loaded during startup. Hardware device drivers are loaded when the driver is opened.[5]

System Global Variables

The Operating System uses **system global variables** to maintain information about the operating environment. Most of these variables contain information of use to the system software, for example:

- Ticks, which contains the number of ticks since system startup. (A tick is 1/60th of a second.)
- MBarHeight, which contains the height of the menu bar.
- Pointers to the heads of various operating system queues.

Other system global variables contain information about the current application, for example:

- ApplZone, which contains the address of the first byte of the active application's memory partition (see Fig 1).

5 Patches are stored as **code resources** of type 'INIT'. Device drivers are stored as code resources of type 'DRVR'. (See Resources, below.)

- ApplLimit, which contains the address of the last byte the active application's heap can expand to include (see Fig 1).

- CurrentA5, which contains the address of the boundary between the active application's global variables and its application parameters (see Fig 1 and Fig 2).

Application Partitions

As shown at Fig 1, an application's **stack** expands downwards towards the **heap**, which expands upwards as necessary during program execution. The ApplLimit global variable marks the upper limit to which the heap can grow. The Memory Manager will never allow the heap to grow beyond ApplLimit.

The Stack

The application stack is used for memory allocation associated with the execution of functions. When an application calls a function, space is automatically allocated on the stack for a **stack frame**, which contains the function's parameters, local variables and return address. Once the call is executed, the local variables and function parameters are popped off the stack.[6]

Unlike the heap, the stack is not bounded by ApplLimit. It is important to understand that the Memory Manager has no way of preventing the stack from growing beyond ApplLimit and possibly encroaching on, and corrupting, the heap.[7]

By default, the stack can grow to 24KB. Accordingly, unless your application uses heavy recursion (one function repeatedly calling itself), you almost certainly will never need to worry about the possibility of stack overflow.[8]

If necessary, you can change the default size of the stack using the function SetApplLimit. When, for example, you call SetApplLimit to *increase* the size of the stack, you are simply reducing the maximum size to which the heap can grow by changing the value in the system global variable ApplLimit (see Fig 1). This gives extra space to the stack at the expense of the heap.

The Heap

The application heap is the area of the application partition in which space is dynamically allocated and released on demand. It contains:

- The application's executable code segments.

- Those of the application's **resources** (see below) that are currently loaded into memory.

- Other dynamically allocated items such as window structures, dialog structures, document data, etc.

Space within the heap is allocated, in blocks, by both direct or indirect calls to the Memory Manager. An indirect allocation arises from a call to a function which itself calls a Memory Manager memory allocation function.

---

6 The C compiler generates the code that creates and deletes stack frames for each function call.

7 However, every sixtieth of a second an Operating System task checks whether the stack has moved into the heap. If it has, the task, known as the **stack sniffer,** generates a system error (System Error 28), which is useful during de-bugging.

8 The reason that recursion increases the risk is that, each time the function calls itself, a *new* copy of that function's parameters and variables is pushed onto the stack.

The A5 World

The **A5 World** contains four kinds of data:

- The application's **jump table**, which contains an entry for each of those of the application's functions that are called by code in another code segment, and which is used by the Segment Manager to determine the address of any externally referenced functions called by a code segment.

- **Application parameters**, which are 32 bytes of memory reserved for use by the Operating System, and of which the first long word is a pointer to the application's QuickDraw global variables (see below).

- The **application global variables**.

- Application **QuickDraw global variables**, which contain information about the application's drawing environment (for example, a pointer to the current graphics port).

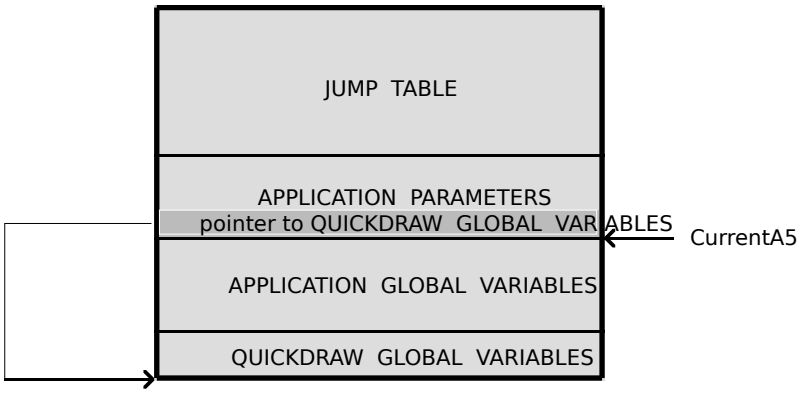


FIG 2 - THE A5 WORLD - 680x0 RUN-TIME ENVI

Fig 2 shows the organisation of this data. Note that the system global variable CurrentA5 points to the boundary between the current application's global variables and its application parameters. The jump table, application parameters, application global variables and QuickDraw global variables are known collectively as the A5 World because the Operating System uses the microprocessor's A5 register to point to that boundary.

Virtual Memory

The Operating System can extend the address space by using part of the available secondary storage (that is, part of the hard disk) to hold portions of applications and data that are not currently needed in RAM. When some of those portions of memory are needed, the Operating System swaps out unneeded parts of applications or data to the secondary storage, thereby making room for the parts that are needed. The secondary storage area is known as **virtual memory**.

Organisation of Memory - PowerPC Run-Time Environment

The organisation of memory in the PowerPC run-time environment is reasonably similar to the organisation of memory in the 680x0 run-time environment in that:

- The system partition occupies the lowest memory address and most of the remaining space is allocated to the Process Manager, which creates a partition for each open application.

- The organisation of an application partition is somewhat the same as that for an application partition in the 680x0 run-time environment. In each application partition, there is a stack and a heap, as well as space for the application's global variables.

The two main differences between 680x0 memory organisation and PowerPC memory organisation concern the location of an application's code section and an application's global variables.

A PowerPC application's executable code and global data are typically stored in a **fragment** container in the application's data fork (see Resources, below). When the application is launched, its **code section** and **data section** are loaded into memory. The data section is loaded into the application's heap. However, the location of the code section varies, depending on whether or not virtual memory is enabled.

Code Section Location - Virtual Memory Off

If virtual memory is not enabled, the code section of an application is loaded into the application heap. The Finder and Process Manager automatically expand your application partition as necessary to hold the code section. The code sections of other fragments are put into part of the Process Manager's heap known as **temporary memory**. If no temporary memory is available, code sections are loaded into the system heap.

Application partitions (including the application's stack, heap, and global variables) are loaded into the Process Manager heap. Code sections of applications and import libraries are loaded either into the Process Manager partition or (less commonly) into the system heap.

Fig 3 illustrates the general organisation of memory when virtual memory is not enabled.

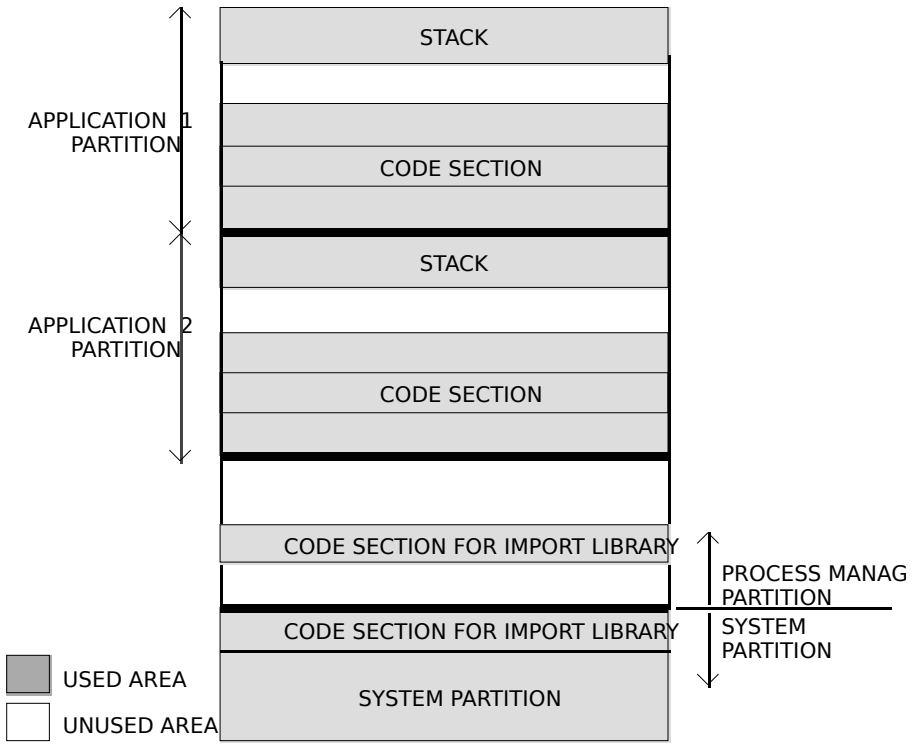


FIG 3 - ORGANIZATION OF MEMORY - POWERPC RUNTIMI VIRTUAL MEMORY IS NOT ENABLED

Code Section Location - Virtual Memory On

If virtual memory is enabled, the Virtual Memory Manager uses a scheme called **file mapping** to map your application's fragment into memory. It uses the data fork of your application as the **paging file** for your application's code section. The entire code fragment is *mapped* into the logical address space, though only the needed portions of the code are actually *loaded* into physical memory.

An advantage of this file mapping methodology is that, when it is time to remove some of your application's code from memory (to page other code and data in), the Virtual Memory Manager does not need to write the pages back to a paging file.[9] Instead, it simply purges the code from the needed pages, because it can always read the file-mapped code back from the paging file (your application's data fork).

Fig 4 illustrates the general organisation of memory when virtual memory is enabled. The virtual addresses occupied by the file-mapped pages of an application's (or an import library's) code are located outside both the system partition and the Process Manager partition. As a result, an application's file-mapped code is never located in the application's heap itself.

Application partitions (including the application's heap, stack, and global variables) are loaded into the Process Manager heap, which is paged to and from the system-wide backing store file. Code sections and import libraries are paged directly from the data fork of the application or import library.

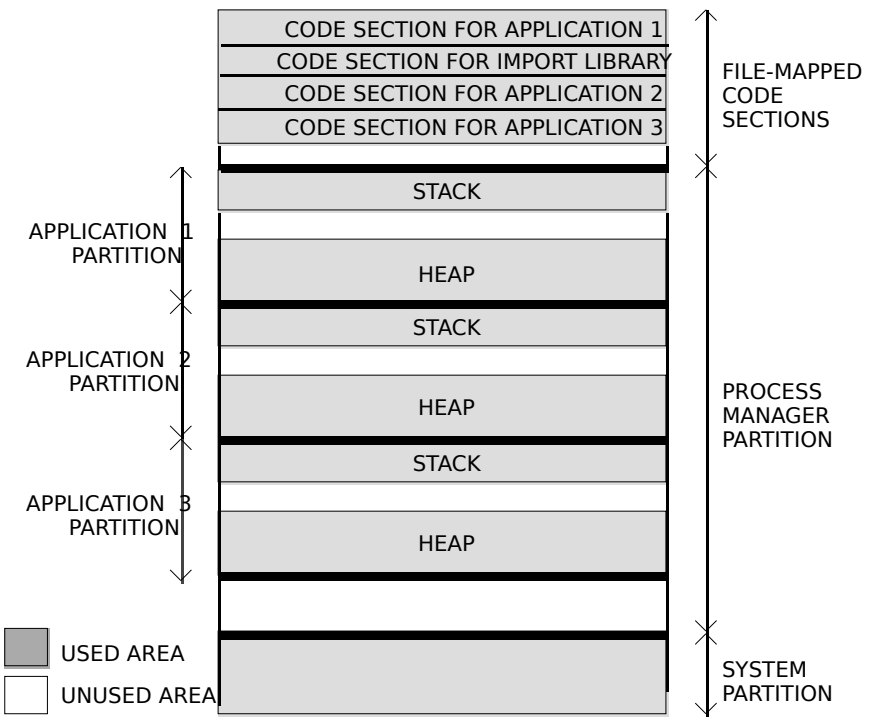


FIG 4 - ORGANIZATION OF MEMORY - POWERPC R ENVIRONMENT - VIRTUAL MEMORY ENABLED

Structure of the System Partition

To support existing 680x0 applications and other software modules which access documented system global variables, the structure of much of the system partition remains unchanged in the PowerPC environment.

Structure of Application Partitions

The organisation of the application partition in the PowerPC environment is substantially simpler than in the 680x0 environment, comprising only a stack and a heap (see Fig 5).

9 In the 680x0 environment, all unused pages of memory are written into a single system-wide backing-store file and re-read from there when needed. This often results in prolonged application launch, because an application's code is loaded into memory and sometimes immediately written out to the backing-store file.

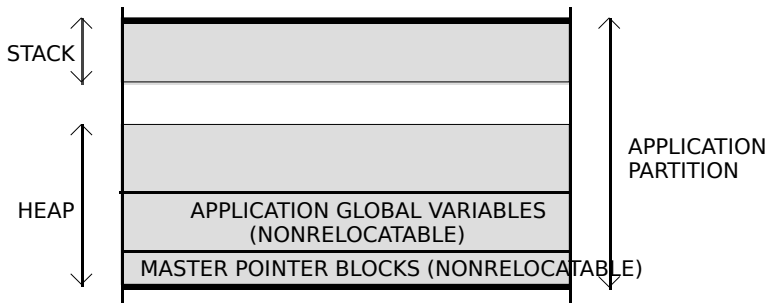


FIG 5 - STRUCTURE OF A POWERPC APPLICATION

Demise of the A5 World

The A5 world which occupies part of a 680x0 application partition is largely absent from the PowerPC environment. The information maintained in the A5 world for 680x0 applications is either not needed by PowerPC applications or is maintained elsewhere (usually in the application heap).

Recall that the A5 world of a 680x0 application contains four kinds of data. The four kinds of data and their fate in the PowerPC environment, are as follows.

- ***Jump Table.*** A 680x0 application's jump table contains an entry for each of the application's routines called by code in another segment. Because the executable code in a PowerPC application is not segmented, there is no need for a jump table in a PowerPC application.

- ***Application Global Variables.*** In PowerPC applications, the application's global variables are part of the fragment's data section, which the Code Fragment Manager loads into the application's heap (see Fig 5). The application's global variables are always located in a single nonrelocatable block.

- ***Application Parameters.*** The application parameters in a 680x0 application occupy 32 bytes, the first four bytes of which are a pointer to the application's QuickDraw global variables. In PowerPC applications, the application parameters are maintained privately by the Operating System.

- ***QuickDraw Global Variables.*** The QuickDraw global variables in a PowerPC application are stored as part of the application's global variables.

The Mini-A5 World

QuickDraw has been ported to native PowerPC code. However, even for applications which have themselves been ported to native PowerPC code, there must be a minimal A5 world to support some non-ported system software which accesses the QuickDraw global variables relative to the application's A5 value. This **mini-A5 world** contains nothing more than a pointer to the application's QuickDraw global variables which, as previously stated, reside in the application's global data section in PowerPC applications. The Process Manager creates a mini-A5 world for each native PowerPC application at application launch time.

Inside the Heap — Nonrelocatable and Relocatable Memory Blocks

An application may use the Memory Manager to allocate two different types of memory blocks: a **nonrelocatable block** and a **relocatable block**.

Nonrelocatable Blocks

Nonrelocatable blocks are blocks whose location in the heap is fixed. In its attempts to avoid heap fragmentation (see below), the Memory Manager allocates nonrelocatable blocks as low in the heap as possible, where necessary moving relocatable blocks upward to make space. Nonrelocatable blocks are referenced using a **pointer** variable of data type Ptr. Ptr is defined as follows:

```
typedef char *Ptr; // A pointer to a signed char.
```

A pointer is simply the address of an arbitrary byte in memory, and a pointer to a nonrelocatable block is simply a pointer to the first byte of that block. Note that, if a copy is made of the pointer variable after the block is created, and since the block cannot be moved, that copy will correctly reference the block until it is disposed of.

The Memory Manager function NewPtr allocates a nonrelocatable block, for example:

```
Ptr   myPointer;
myPointer = NewPtr(sizeof(WindowRecord));
```

Nonrelocatable blocks are disposed of by a call to DisposePtr.[10]

Unlike relocatable blocks, there are only five things that your application can do with a nonrelocatable block: create it; obtain its size; resize it; find which heap zone owns it; dispose of it.

Relocatable Blocks

Relocatable blocks are blocks which can be moved within the heap — for example, during heap compaction operations (see below). To reference relocatable blocks, the Memory Manager uses **double indirection**, that is, the Memory Manager keeps track of a nonrelocatable block with a **master pointer**, which is itself part of a nonrelocatable **master pointer block** in the application heap. When the Memory Manager moves a relocatable block, it updates the master pointer so that it always contains the address of the relocatable block.

The Memory Manager allocates one master pointer block, which contains 64 master pointers, for the application at launch time. This block is located at the very bottom of the application heap. MoreMasters may be called by the application to allocate additional master pointer blocks. To ensure that these additional (nonrelocatable) blocks are allocated as low in the heap as possible, the calls to MoreMasters should be made at the beginning of the program.[11]

Relocatable blocks are referenced using a **handle** variable of data type Handle. A handle contains the address of a master pointer, as illustrated at Fig 6. Handle is defined as follows:

```
typedef Ptr *Handle; // A pointer to a master pointer.
```

The Memory Manager function NewHandle allocates a relocatable block, for example:

```
Handle myHandle;
myHandle = NewHandle(sizeof(myDataStructure));
```

---

10 Many system software functions can be accessed using more than one spelling of the function's name, depending on the header files supported by the development environment. For example, several years ago, the name for this function was DisposPtr.

11 If these calls are not made, the Memory Manager will nonetheless automatically allocate additional blocks during application execution if required. However, since master pointer blocks are nonrelocatable, such allocation, which will not be at the bottom of the heap, is a possible cause of heap fragmentation. MoreMasters should thus be called enough times at the beginning of the program to ensure that the Memory Manager never needs to call it for you. For example, if your application never allocates more than 300 relocatable blocks in its heap, then five calls to MoreMasters should be enough. (You can empirically determine how many times to call MoreMasters by using a low-level debugger.)

A relocatable block can be disposed of by a call to DisposeHandle. Note, however, that the Memory Manager does not change the value of any handle variables that previously referenced that deallocated block. Instead, those variables still hold the address of what was once the relocatable block's master pointer. If you accidentally use a handle to a block you have already disposed of, your application could crash or you could get garbled data. You can avoid these problems by assigning the value NULL to the handle variable after you dispose of the relocatable block.

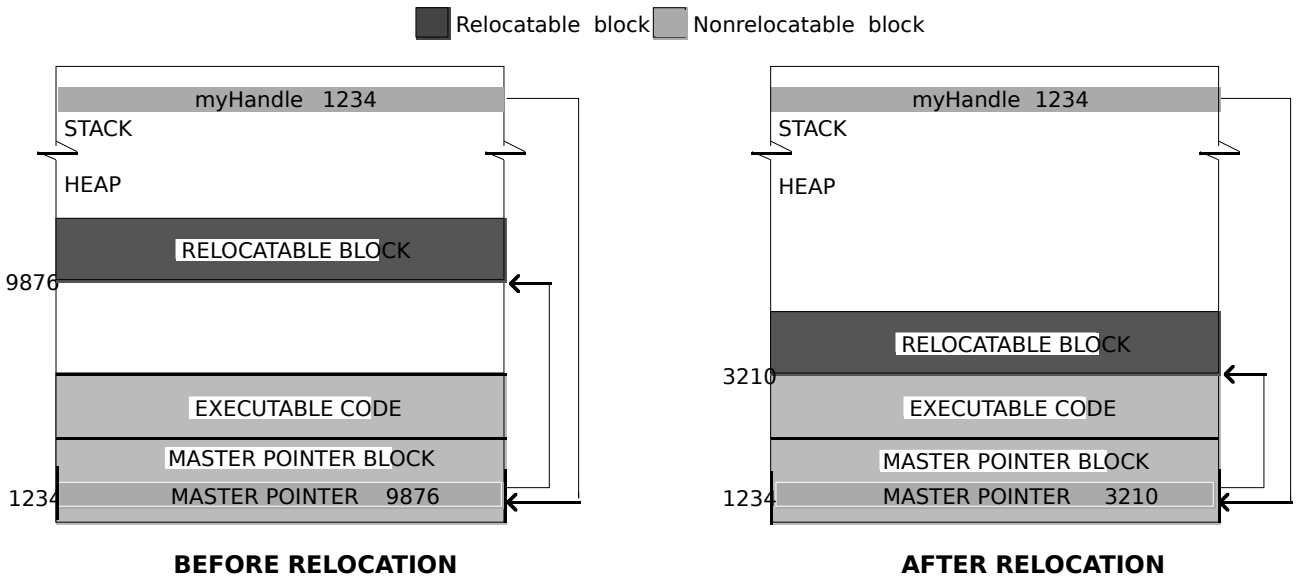


FIG 6 - A HANDLE TO A RELOCATABLE BLOCK

Heap Fragmentation, Compaction, and Purging

The continuous allocation and release of memory blocks which occurs during an application's execution can result in a condition called **heap fragmentation**. The heap is said to be fragmented when nonrelocatable blocks or **locked relocatable blocks** (see below) are scattered about the heap, leaving "holes" of memory between those blocks.

The Memory Manager continuously attempts to create more contiguous free memory space through an operation known as **heap compaction**, which involves moving all relocatable blocks as low in the heap as possible. However, because the Memory Manager cannot move relocatable blocks "around" nonrelocatable blocks and locked relocatable blocks, such blocks act like log-jams if there is free space below them. In this situation, the Memory Manager may not be able to satisfy a new memory allocation request because, although there may be enough total free memory space, that space is broken up into small non-contiguous blocks.

Heap fragmentation would not occur if all memory blocks allocated by the application were free to move during heap compaction. However, there are two types of memory block which are not free to move: nonrelocatable blocks and relocatable blocks which have been temporarily locked in place.

Locking and Unlocking Relocatable Blocks

Despite the potential of such action to inhibit the Memory Manager's heap compaction activities, it is nonetheless necessary to lock relocatable blocks in place in certain circumstances.

For example, suppose you dereference a handle to obtain a pointer (that is, a *copy* of the master pointer) to a relocatable block and, for the sake of increased speed[12], use that pointer within a loop to read or write data to or from the block. If, within that loop, you

12 Accessing a relocatable block by double indirection (that is, through its handle) instead of by single indirection (ie, through its master pointer) requires an extra memory reference.

call a function which has the potential to move memory, and if that function actually causes the relocatable block to be moved, the master pointer will be correctly updated but your copy (the pointer) will not. The net result is that your pointer no longer points to the data and becomes what is known as a **dangling pointer**. This situation is illustrated at Fig 7.

The documentation for system software functions indicates whether a particular function has the potential to move memory. Generally, any function that allocates space from the application heap has this potential. If such a function is not called in a section of code, you can safely assume that all blocks will remain stationary while that code executes.

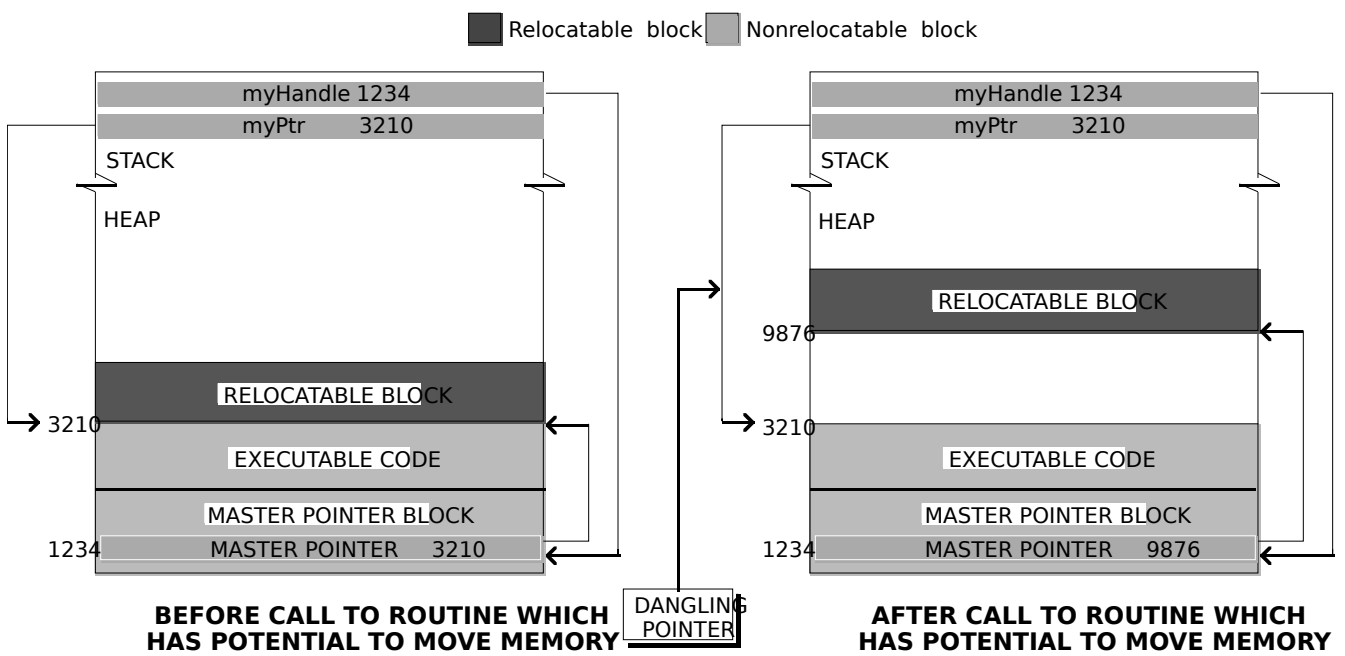


FIG 7 - A DANGLING POINTER

Relocatable blocks may be locked and unlocked using HLock and HUnlock. The following example illustrates the use of these functions.

```
typedef struct
{
    short  intArray[1000];
    char       ch;
} Structure, *StructurePointer, **StructureHandle;

void  myFunction(void)
{
    StructureHandle         theHdl;
    StructurePointer        thePtr;
    short      count;

    theHdl = (StructureHandle) NewHandle(sizeof(Structure));

    HLock(theHdl);                              // Lock the relocatable block ...
    thePtr = *theHdl;                           // because the handle has been dereferenced ...

    for(count=0;count<1000;count++)
    {
        (*thePtr).intArray[count] = 0;          // and used in this loop ...
        DrawChar((char)'A');                    // which calls a function which could cause
    }                                           // the relocatable block to be moved.

    HUnlock(theHdl);                            // On loop exit, unlock the relocatable block.
}
```

Moving Relocatable Blocks High

The potential for a locked relocatable block to contribute to heap fragmentation may be avoided by moving the block to the top of the heap before locking it. This should be done if new nonrelocatable blocks are to be allocated while the relocatable block in question is locked.

MoveHHi is used to move relocatable blocks to the top of the heap. HLockHi is used to move relocatable blocks to the top of the heap and then lock them. Be aware, however, that MoveHHi and HLockHi cannot move a block to the top of the heap if a nonrelocatable block or locked relocatable block is located between its current location and the top of the heap. In this situation, the block will be moved to a location immediately below the nonrelocatable block or locked relocatable block.

Purging and Reallocating Relocatable Blocks

In addition to compacting the heap in order to satisfy a memory allocation request, the Memory Manager may **purge** unlocked relocatable memory blocks which have been made purgeable.

HPurge and HNoPurge change a relocatable block from unpurgeable to purgeable and vice versa. When you make a relocatable block purgeable, your program should subsequently check the handle to that block before using it if calls are made to functions which could move or purge memory.

If the handle's master pointer is set to NULL, then the Operating System has purged its block. To use the information formerly in the block, space must then be reallocated for it and its contents must be reconstructed.

Effect of a Relocatable Block's Attributes

Two **attributes** of a relocatable block are whether the block is currently locked/unlocked or purgeable/non-purgeable. These attributes are stored in bits in the block's master pointer **tag byte**[13]. The following summarises the effect of these attributes on the Memory Manager's ability to move and/or purge a relocatable block:

Tag Byte Indicates Block Is:		The Memory Manager Can:	
Locked	**Purgeable**	**Move The Block**	**Purge the Block**
NO	NO	**YES**	NO
NO	**YES**	**YES**	**YES**
YES	NO	NO	NO
YES	**YES**	NO	NO

Note that a relocatable block created by a call to NewHandle is created initially unlocked and unpurgeable, and that locking a relocatable block will also make it unpurgeable if it is currently purgeable.

Avoiding Heap Fragmentation

The ideal heap is one with all nonrelocatable blocks at the bottom of the heap, all unlocked relocatable blocks above that, free space above that, and all relocatable blocks which must be locked for significant periods at the top of the heap. This ideal can be approached, and significant heap fragmentation avoided, by adherence to the following rules:

- At the beginning of the program, call MaxApplZone to expand the heap immediately to ApplLimit. (If MaxApplZone is not called, the Memory Manager gradually expands the heap towards ApplLimit as memory needs dictate. This gradual expansion can result in significant heap fragmentation if relocatable blocks have previously been moved to the previous top of the heap and locked.)[14]

13 The tag byte is the high byte of a master pointer. If Bit 5 of the tag byte is set, the block is a resource block (see below). If Bit 6 is set, the block is purgeable. If Bit 7 is set, the block is locked.

- At the beginning of the program, call MoreMasters enough times to allocate all of the (nonrelocatable) master pointer blocks required during execution.

- Allocate all other required nonrelocatable blocks at the beginning of the application's execution.

- Avoid disposing of, and then reallocating, nonrelocatable blocks during the application's execution.

- When allocating relocatable blocks that will need to be locked for long periods of time, use ReserveMem at the beginning of the program to reserve memory for them as close to the bottom of the heap as possible, and lock the blocks immediately after allocating them.

- If a relocatable block is to be locked for a short period of time and nonrelocatable blocks are to be allocated while it is locked, call MoveHHi to move the block to the top of the heap and then lock it. When the block no longer needs to be locked, unlock it.

Also bear in mind that, in memory management terms, a relocatable block that is always locked is worse than a nonrelocatable block in that nonrelocatable blocks are always allocated as low in the heap as possible, whereas a relocatable block is allocated wherever the Memory Manager finds it convenient.

Master Pointer Tag Byte - HGetState and HSetState

There are certain circumstances where you will want to save, and later restore, the current value of a relocatable block's master pointer tag byte. Consider the following example, which involves three of an imaginary application's functions, namely, Function A, Function B, and Function C:

- Function A creates a relocatable block. For reasons of its own, Function A locks the block before executing a few lines of code. Function A then calls Function C, passing the handle to that function as a formal parameter.

- Function B also calls Function C at some point, passing the relocatable block's handle to it as a formal parameter. The difference in this instance is that, due to certain machinations in other areas of the application, the block is unlocked when the call to Function C is made.

- Function C, for reasons of its own, needs to ensure that the block is locked before executing a few lines of code, so it makes a call to HLock. Those lines executed, Function C then unlocks the block before returning to the calling function. This will not be of great concern if the return is to Function B, which expects the block to be still unlocked. However, if the return is to Function A, and if Function A now executes some lines of code which assume that the block is still locked, disaster could strike.

This is where the Memory Manager functions HGetState and HSetState come in. The sequence of events in Function C should have been as follows:

```
SInt8 theTagByte;
...
theTagByte = HGetState(myHandle);          // Whatever the current state is, save it.
HLock(myHandle);                           // Redundant if Function A is calling, but no harm.

(Bulk of the Function C code, which requires handle to be locked.)

HSetState(myHandle,theTagByte)             // Leave it the way it was found.  (It could have
```

14 Another reason for calling MaxApplZone at the beginning of the program is that the number of purgeable memory blocks that may need to be purged by the Memory Manager to satisfy a new memory request is reduced. (The Memory Manager expands the heap to fulfill a memory request only after it has exhausted other methods of obtaining the required amount of space, including compacting the heap and purging blocks marked as purgeable.) Also, since calling MaxApplZone means that the heap is expanded only once during the application's execution, memory allocation operations can be significantly faster.

```
                                                    // been locked.  It could have been unlocked.)
    return;
```

This is an example of a of what might be called a “well-mannered function”. It is an example of a rule that you may wish to apply whenever you write a function that takes a handle to a relocatable block as a formal parameter: If that function calls HLock, make sure that it leaves the block's tag byte (and thus the locked/unlocked bit) in the condition in which it found it.[15]

Addressing Modes

Early versions of the system software used 24-bit addressing, where the upper eight bits of memory addresses were ignored or used as flag bits. 24-bit addressing limits the address space to 16MB, 8MB of which is reserved for I/O space, ROM and slot space. The largest contiguous program address space under 24-bit addressing is thus 8MB.

Later versions of the Memory Manager (specifically, the 32-bit Memory Manager) support 32-bit addressing, under which the maximum program address space is 1GB.

For compatibility reasons, systems with a 32-bit Memory Manager also contain a 24-bit Memory Manager. In order for an application to work when the machine is using 32-bit addressing, it must be **32-bit clean**. Some applications are not 32-bit clean because they use flag bits in master pointers and manipulate those bits directly (for example, to mark the associated memory blocks as locked or purgeable) instead of using Memory Manager functions to achieve the same results. You must avoid such practices if your application is to be 32-bit clean.

Memory Leaks

When you have no further use for a block of memory, you ordinarily return that memory to the Memory Manager by calling DisposePtr or DisposeHandle (or ReleaseResource (see below)). In certain circumstances, not disposing of a block which is no longer required can result in what is known as a **memory leak**.

Memory leaks can have unfortunate consequences for your application. For example, consider the following function:

```
void  theFunction(void)
{
    PtrthePointer;
    OSErr     osError;

    thePointer = NewPtr(10000);
    if(MemError() == memFullErr)
        doErrorAlert(eOutOfMemory);

    // The nonrelocatable block is used for some temporary purpose here, but is not
    // disposed of before the function returns.
}
```

When theFunction returns, the 10000-byte nonrelocatable block will still exist (even though, incidentally, the local variable which previously pointed to it will not). Thus a large nonrelocatable block for which you have no further use remains in memory (at what is now, incidentally, an unknown location). If theFunction is called several more times, a new nonrelocatable block will be created by each call and the size of the memory leak will grow, perhaps eventually causing MemErr to return memFullErr. In this way, memory leaks can bring you application to a standstill and may, in some circumstances, cause it to crash.[16]

---

15 Of course, this save/restore precaution will not really be necessary if you are absolutely certain that the block in question will be in a particular state (locked or unlocked) every time Function C is called. But there is nothing wrong with a little coding overkill to protect yourself from, for example, some future source code modifications which may add other functions which call Function C, and which may assume that the block's attributes will be handed back in the condition in which Function C found them.

Memory Manager Errors

The low-memory address 0x0220, which is represented by the symbolic name MemErr, contains the error code resulting from the last call to a Memory Manager function. This error code may be retrieved by calling the function MemError. Some of the error codes which may be returned by MemError are as follows:

Error Code	Constant	Description
0	noErr	No error occurred.
-108	memFullErr	No room in heap.
-109	nilHandleErr	Illegal operation on a NULL handle.

Resources

In order to meet various requirements of the system software, your application must provide its own **resources**, for example, resources which describe the application's user interface elements such as menus, windows, controls, dialog boxes and icons. In addition, the system software itself provides a number of resources (for example, fonts, patterns, icons, etc.) which may be used by your application.

The concept of resources reflects the fact that, in the Macintosh environment, inter-mixing code and data in a program is not encouraged. For example, it is usual practise to separate changeable data, such as message strings which may need to be changed in a foreign-language version of the application, from the application's code. All that is required in such a case is to create a resource containing the foreign language version of the message strings. There is thus no necessity to change and recompile the source code in order to produce a foreign-language version of the application.

The subject of resources is closely related to the subject of files. A brief digression into the world of files is thus necessary.

About Files — The Data Fork and the Resource Fork

On the Macintosh, a **file** is a named, ordered sequence of bytes stored on a volume and divided into two forks:

- ***The Data Fork.*** The data fork typically contains data created by the user.
- ***The Resource Fork***. The resource fork of a file contains **resources**, which are collections of data of a defined structure and type.

All Macintosh files contain both a resource fork and a data fork, although one or both of these forks may be empty. Note that the resource fork of a file is also called a **resource file**, because in some respects you can treat it as if it were a separate file.

The resource fork of a document file contains any document-specific resources, such as the size and location of the document's window when the document was last closed. The resource fork of an application file includes the application's executable 680x0 code and, typically, resources which describe the application's windows, menus, controls, dialog boxes, icons, etc. Fig 8 illustrates the typical contents of the data and resource forks of an application file and a document file.

16 The dynamic memory inspection tool ZoneRanger, which is included with Metrowerks CodeWarrior, can be used to check your application for memory leaks.

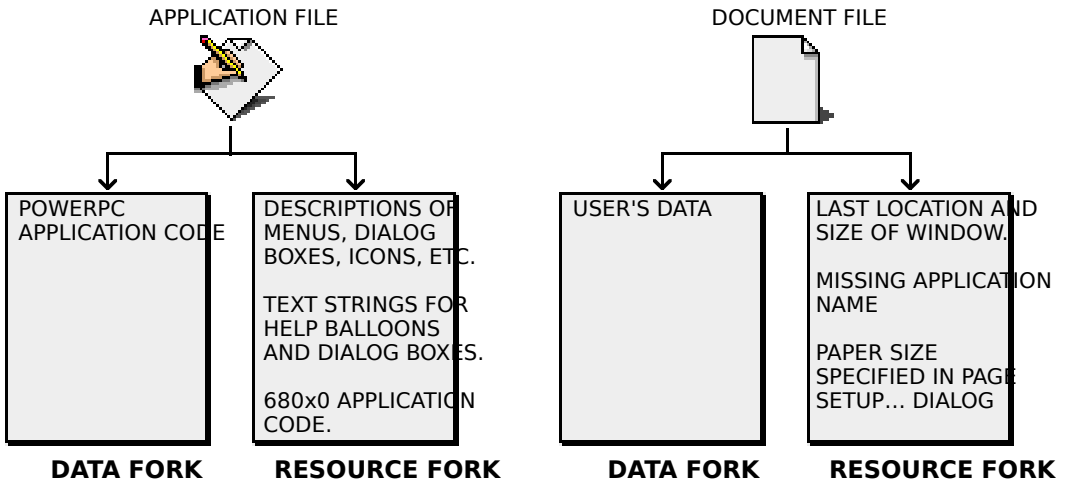


FIG 8 - TYPICAL CONTENTS OF DATA FORKS AND RESOURCE FORKS IN APPLICA

The data fork can contain any kind of data organised in any fashion. Your application can store data in the data fork of a document file in whatever way it deems appropriate, but it needs to keep track of the exact byte location of each particular piece of saved data in order to be able to access that data when required. The resource fork, on the other hand, is highly structured. As will be seen, all resource forks contain a map which, amongst other things, lists the location of all resources in the resource fork. This greatly simplifies the task of accessing those resources.

Resources and the Application

During its execution, an application may read resources from:

- The application's resource file, which is opened automatically when the application is launched.

- The System file, which is opened by the Operating System at startup and which contains resources which are shared by all applications (for example, fonts, icons, sounds, etc.) and resources which applications may use to help present the standard user interface.

- Other resource files, such as a preferences file in the Preferences folder holding the user's application-specific preferences, or the resource fork of a document file, which might define certain document-specific preferences.

The Resource Manager provides functions which allow your application to read in these resources and, in addition, to create, delete, open, modify and write resources in, from and to any Macintosh file. The following, however, is concerned only with creating resources for the application's resource file and with reading in **standard resources** from the application and System files. Other aspects of resources, including **custom resources** and resources in files other than the application and System files, are addressed at Chapter 17 — More on Resources.

Resource Types and Resource IDs

An application refers to a resource by passing the Resource Manager a **resource specification**, which consists of the **resource type** and a **resource ID**:

- ***Resource Type.*** A resource type is specified by any sequence of four alphanumeric characters, including the space character, which uniquely identifies a specific type of resource. Both uppercase and lowercase characters are used. Some of the standard resource types defined by the system software are as follows:

Type	*Description*	*Type*	*Description*

'ALRT'	Alert box template.	'CODE'	Application code segment.
'DITL'	Item list in dialog or alert box.	'DLOG'	Dialog box template.
'FONT'	Bitmapped font.	'ICON'	Large black-and-white icon.
'MBAR'	Menu bar.	'PAT '	Pattern. (Space character is required.)
'PICT'	QuickDraw picture.	'SIZE'	Size of application's partition and other info.
'STR#'	String list.	'WIND'	Window template.
'snd '	Sound. (Space character is required.)	'sfnt'	Outline font.

You can also create your own custom resource types if your application needs resources other than the standard types. An example would be a custom resource type for application-specific preferences stored in a preferences file.[17]

- ***Resource ID.*** A resource ID identifies a specific resource of a given type by number. System resource IDs range from -32768 to 127. In general, resource IDs from 128 to 32767 are available for resources that you create yourself, although the numbers you can use for some types of resources (for example, font families) are more restricted. An application's **definition functions** (see below) should use IDs between 128 and 4095.

Creating a Resource

At the very least, you will need to create resources for the standard user interface elements used by your application. You typically define the user interface elements in resources and then use Menu Manager, Window Manager, Dialog Manager or Control Manager functions to create these elements, based on their resource descriptions, as needed.

You can create resource descriptions using a resource editor such as Resorcerer (which uses the familiar point-and-click approach), or you can provide a textual, formal description of resources in a file and then use a resource compiler, such as Rez, to compile the description into a resource.[18] An example of a resource definition for a window in Rez input format is as follows:

```
resource 'WIND' (128, preload, purgeable)
{
    {64,60,314,460},                /* Window rectangle. (Initial window size and location.) */
    kWindowDocumentProc,            /* Window definition ID. */
    invisible,                      /* Window is initially invisible. */
    goAway,                         /* Window has a close box. */
    0x0,                            /* Reference constant. */
    "untitled",                     /* Window title. */
    staggerParentWindowScreen       /* Optional positioning specification. */
};
```

The structure of the compiled 'WIND' resource is shown at Fig 9.

17 When choosing the characters to identify your custom resource types, note that Apple reserves for its own use resource types consisting entirely of lowercase characters and special symbols. Your custom resource types should therefore contain at least one uppercase character.

18 Macintosh C assumes the use of Resorcerer, and all demonstration program reources were created using Resorcerer.

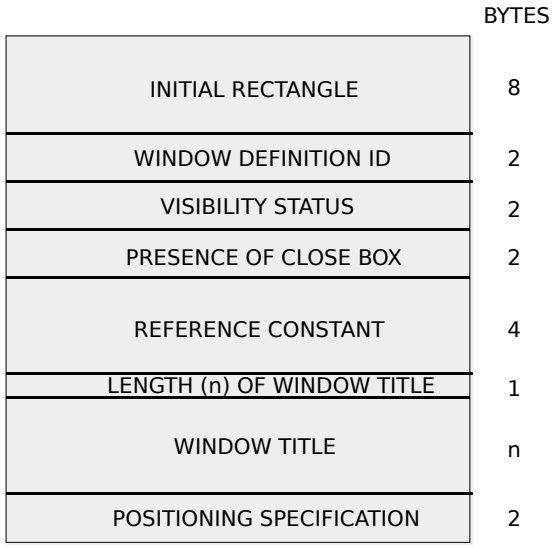


FIG 9 - STRUCTURE OF A COMPILED WINDOW ('

Resource Attributes

Note the words preload and purgeable in the preceding 'WIND' resource definition. These are constants representing **resource attributes**, which are flags which tell the Resource Manager how to handle the resource. Resource attributes are described by bits in the low-order byte of an integer value:

Bit	*Constant*	*Description*
1	resChanged	Resource has been changed.
2	resPreload	Resource is to be read into memory immediately after the resource fork is opened.
3	resProtected	Application cannot change the resource ID, modify the resource's contents or remove the resource from the resource fork.
4	resLocked	Relocatable block occupied by the resource is to be locked. (Overrides the resPurgeable attribute.)
5	resPurgeable	Relocatable block occupied by the resource is to be purgeable.
6	resSysHeap	Read the resource into the system heap rather than the application heap.

Note that, if both the resPreload and the resLocked attributes are set, the Resource Manager loads the resource as low as possible in the heap.

Resources Which Must Be Unpurgeable. Some resources must *not* be made purgeable. For example, the Menu Manager expects menu resources to remain in memory at all times.

Resources Which May Be Purgeable. Other resources, such as those relating to windows, controls, and dialog boxes, do not have to remain in memory once the corresponding user interface element has been created. You may therefore set the purgeable attribute for those kinds of resources if you so desire. The following considerations apply to the decision as to whether to make a resource purgeable or unpurgeable:

- The concept of purgeable resources dates back to the time when RAM was limited and programmers had to be very careful about allowing resources which were not in use to continue to occupy precious memory. Nowadays, however, RAM is not so limited, and programmers need not be overly concerned about, say, a few 'DLOG' resources (24 bytes each) remaining in memory when they are not required.

- Some resources (for example, large 'PICT' resources and 'snd ' resources) do require a lot of memory, even by today's standards. Accordingly, such resources should generally be made purgeable.

- As will be seen, there are certain hazards associated with the use of purgeable resources. These hazards must be negated by careful programming involving additional lines of code.

Given these considerations, a sound policy would be to make all small and basic resources unpurgeable and set the resPurgeable attribute only in the case of comparatively large resources which are not required to remain permanently in memory.

Template Resources and Definition Resources

The 'WIND' resource defined above is an example of a **template resource**. A template resource defines the characteristics of a desktop object, in this case a window's size, location, etc., and the **window definition function** (specified by the constant kWindowDocumentProc) to be used to draw it. Definition functions, which determine the look and behaviour of a desktop object, are executable code segments contained within another kind of resource called a **definition resource**.

The definition function specified by the constant kWindowDocumentProc is contained within the 'WDEF' resource with ID 64 in the System file. Note that it is possible to write your own custom window definition function (and, indeed, custom definition functions for other desktop objects such as menus), store it in a 'WDEF' resource in you application file, and specify it in the relevant field of your 'WIND' resource definitions.

Resources in Action

The Resource Map

Your application file's resource fork contains, in addition to the resources you have created for your application, an automatically created **resource map**. The resource map contains entries for each resource in the resource fork.

When your application is launched, the system first gets the Memory Manager to create the application heap and allocate a block of master pointers at the bottom of the heap. The Resource Manager then opens your application file's resource fork and reads in the resource map, followed by those resources which have the resPreload attribute set.

The handles to the resources which have been loaded are stored in the resource map in memory. The following is a diagrammatic representation of a simple resource map in memory immediately after the resource map, together with those resources with the preload attribute set, have been loaded.

Type	ID	Attributes			Handle
		Preload	Lock	Purgeable	
CODE	1	•	•		1234
CODE	2		•		NULL
MENU	128	•			123C
WIND	128			•	NULL
PICT	128			•	NULL
PICT	129			•	NULL

Note that the handle entry in the resource map contains NULL for those resources which have not yet been loaded. Note also that this handle entry is filled in only when a resource is loaded for the first time, and that that entry remains even if a purgeable resource is later purged by the Memory Manager.

Reading in Non-Preloaded Resources

Some system software managers use the Resource Manager to read in resources for you. Using the 'WIND' resource listed in the above resource map as an example, when the Window Manager function GetNewCWindow is called to create a new window (specifying 128 as the resource ID), GetNewCWindow, in turn, calls the Resource Manager function GetResource. GetResource loads the resource (assuming that it is not currently in memory), returns the handle to GetNewCWindow, and copies the handle to the appropriate entry in the resource map. This is an example of an indirect call to the Resource Manager.

Other resources are read in by direct calls to the Resource Manager. For example, the 'PICT' resources listed in the above example resource map would be read in by calling another of the Get... family of resource-getting functions directly, for example:

```
#define rPicture1 128
#define rPicture2 129
...
PicHandle pic1Hdl;
PicHandle pic2Hdl;
...
pic1Hdl = GetPicture(rPicture1);
pic2Hdl = GetPicture(rPicture2);
```

Once again, and assuming that the resources have not previously been loaded, the handle returned by each GetPicture call is copied to the appropriate entry in the resource map.

Purgeable Resources

When a resource which has the resPurgeable attribute set has been loaded for the first time, the handle to that resource is copied to the appropriate entry in the resource map in the normal way. If the Memory Manager later purges the resource, the master pointer pointing to that resource is set to NULL by the Memory Manager but the handle entry in the resource map remains. This creates what is known as an **empty handle**.

If the application subsequently calls up the resource, the Resource Manager first checks the resource map handle entry to determine whether the resource has ever been loaded (and thus whether a master pointer exists for the resource). If the resource map indicates that the resource has never been loaded, the Resource Manager loads the resource, returns its handle to the calling function, and copies the handle to the resource map.

If, on the other hand, the resource map indicates that the resource has previously been loaded (that is, the handle entry in the resource map contains the address of a master pointer), the Resource Manager checks the master pointer. If the master pointer contains NULL, the Resource Manager knows that the resource has been purged, so it reloads the resource and updates the master pointer. Having satisfied itself that the resource is in memory, the Resource Manager returns the resource's handle to the application.

Problems with Purgeable Resources

Using purgeable resources optimises heap space; however, misuse of purgeable resources can crash an application. For example, consider the following code example, which loads two purgeable 'PICT' resources and then uses the drawing instructions contained in those resources to draw each picture.

```
pic1Hdl = GetPicture(rPicture1);          // Load first 'PICT' resource.
pic2Hdl = GetPicture(rPicture2);          // Load second 'PICT' resource.
if(pic1Hdl)                               // If the handle to first resource is not NULL ...
    DrawPicture(pic1Hdl,&destRect);       // ... draw the second picture.
if(pic2Hdl)                               // If the handle to second resource is not NULL
    DrawPicture(pic2Hdl,&destRect);       // ... draw the second picture.
```

GetPicture is one of the many functions that can cause memory to move. When memory is moved, the Memory Manager may purge memory to obtain more heap space. If heap space is extremely limited at the time of the second call to GetPicture, the first resource will

be purged by the Memory Manager, which will set the master pointer to the first resource to NULL to reflect this condition. The variable pic1Hdl will now contain an empty handle. Passing an empty handle to DrawPicture just about guarantees a system crash.

There is a second problem with this code. Like GetPicture, DrawPicture also has the potential to move memory blocks. If the second call to GetPicture did not result in the first resource being purged, the possibility remains that it will be purged while it is being used (that is, during the execution of the DrawPicture function).

To avoid such problems when using purgeable resources, you should observe these steps:

- Get (that is, load) the resource only when it is needed.
- Immediately make the resource unpurgeable.
- Use the resource immediately after making it unpurgeable.
- Immediately after using the resource, make it purgeable.

The following revised version of the above code demonstrates this approach:

```
pic1Hdl = GetPicture(rPicture1);            // Load first 'PICT' resource.
if(pic1Hdl)                                 // If the resource was successfully loaded ...
{
   HNoPurge((Handle) pic1Hdl);              // make the resource unpurgeable ...
   DrawPicture(pic1Hdl,&destRect);          // draw the first picture ...
   HPurge((Handle) pic1Hdl);                // and make the resource purgeable again.
}

pic2Hdl = GetResource(rPicture2);           // Repeat for the second 'PICT' resource.
if(pic2Hdl)
{
   HNoPurge((Handle) pic2Hdl );
   DrawPicture(pic2Hdl,&destRect);
   HPurge((Handle) pic2Hdl );
}
```

Note that this procedure only applies when you use functions which get resources directly (for example GetResource, GetPicture, etc.). It is not required when you call GetResource indirectly (for example, when you call the Window Manager function GetNewWindow) because functions like GetNewWindow know how to treat purgeable resources properly.

Note also that LoadResource may be used to ensure that a previously loaded, but purgeable, resource is in memory before an attempt is made to use it. If the specified resource is not in memory, LoadResource will load it. The essential difference between LoadResource and the Get... family of resource-getting functions is that the latter *return* a handle to the resource (loading the resource if necessary), whereas LoadResource *takes* a handle to a resource as a parameter and loads the resource if necessary.

Releasing Resources

When you have finished using a resource loaded by a function which gets resources directly, you should call the appropriate function to release the memory associated with that resource. For example, ReleaseResource is used in the case of generic handles obtained with the GetResource function. ReleaseResource frees up all the memory occupied by the resource and sets the resource's handle in the resource map to NULL.

You do not need to be concerned with explicitly releasing resources loaded indirectly (for example, by a call to GetNewCWindow). Using the case of a window resource template as an example, the sequence of events following a call to GetNewCWindow is as follows:

- GetNewCWindow calls GetResource to read in the window resource template whose ID is specified in the GetNewCWindow call.

- A relocatable block is created for the template resource and marked as purgeable, as specified by the resource's attributes. (You should always specify window template resources as purgeable.)
- The window template's block is then temporarily marked as unpurgeable while:
  - A nonrelocatable block is created for a data structure known as a window structure.
  - Data is copied from the resource template into certain fields in the window structure.
- The window template's block is then marked as purgeable.

Resource Manager Errors

The low-memory address 0x0A60, which is represented by the symbolic name ResErr, contains the error code resulting from the last call to a Resource Manager function. This error code may be retrieved by calling the function ResError. Some of the error codes which may be returned by ResError are as follows:

Error Code	Constant	Description
0	noErr	No error occurred.
-192	resNotFound	Resource not found.
-193	resFNotFound	Resource file not found.

Main Memory Manager Data Types and Functions

Data Types

```
typedef char    *Ptr;       // Pointer to nonrelocatable block.
typedef Ptr     *Handle;    // Handle to relocatable block.
typedef long    Size;       // Size of a block in bytes.
```

Functions

Setting Up the Application Heap

```
void    MaxApplZone(void);
void    MoreMasters(void);
Ptr     GetApplLimit(void);
void    SetApplLimit(void *zoneLimit);
```

Allocating and Releasing NonRelocatable Blocks of Memory

```
Ptr     NewPtr(Size byteCount);
Ptr     NewPtrClear(Size byteCount);
Ptr     NewPtrSys(Size byteCount);
Ptr     NewPtrSysClear(Size byteCount);
void    DisposePtr(Ptr p);
```

Allocating and Releasing Relocatable Blocks of Memory

```
Handle  NewHandle(Size byteCount);
Handle  NewHandleClear(Size byteCount);
Handle  NewHandleSys(Size byteCount);
Handle  NewHandleSysClear(Size byteCount);
Handle  NewEmptyHandle(void);
Handle  NewEmptyHandleSys(void);
void    DisposeHandle(Handle h);
```

Changing the Sizes of Nonrelocatable and Relocatable Blocks

```
Size        GetPtrSize(Ptr p);
void        SetPtrSize(Ptr p,Size newSize);
Size        GetHandleSize(Handle h);
void        SetHandleSize(Handle h,Size newSize);
```

Setting the Properties of Relocatable Blocks

```
void        HLock(Handle h);
void        HUnlock(Handle h);
void        HPurge(Handle h);
void        HNoPurge(Handle h);
SInt8       HGetState(Handle h);
void        HSetState(Handle h,SInt8 flags);
```

Managing Relocatable Blocks

```
void        EmptyHandle(Handle h);
void        ReallocateHandle(Handle h,Size byteCount);
Handle      RecoverHandle(Ptr p);
void        ReserveMem(Size cbNeeded);
void        ReserveMemSys(Size cbNeeded);
void        MoveHHi(Handle h);
void        HLockHi(Handle h);
```

Manipulating Blocks of Memory

```
void        BlockMove(const void *srcPtr,void *destPtr,Size byteCount);
void        BlockMoveData(const void *srcPtr,void *destPtr,Size byteCount);
OSErr       PtrToHand(const void *srcPtr,Handle *dstHndl,long size);
OSErr       PtrToXHand(const void *srcPtr,Handle dstHndl,long size);
OSErr       HandToHand(Handle *theHndl);
OSErr       HandAndHand(Handle hand1,Handle hand2);
OSErr       PtrAndHand(const void *ptr1,Handle hand2,long size);
```

Accessing Memory Conditions

```
long        FreeMem(void);
long        FreeMemSys(void);
long        MaxBlock(void);
long        MaxBlockSys(void);
void        PurgeSpace(long *total,long *contig);
long        StackSpace(void);
```

Freeing Memory

```
Size        CompactMem(Size cbNeeded);
Size        CompactMemSys(Size cbNeeded);
void        PurgeMem(Size cbNeeded);
void        PurgeMemSys(Size cbNeeded);
Size        MaxMem(size *grow);
Size        MaxMemSys(size *grow);
```

Allocating Temporary Memory

```
Handle      TempNewHandle(Size logicalSize,OSErr *resultCode);
long        TempFreeMem(void);
Size        TempMaxMem(Size *grow);
```

Checking for Errors

```
OSErr       MemError(void);
```

Main Resource Manager Constants, Data Types, and Functions

Constants

Resource Attributes

```
resSysHeap      = 64    System or application heap?
resPurgeable    = 32    Purgeable resource?
resLocked       = 16    Load it in locked?
resProtected    = 8     Protected?
resPreload      = 4     Load in on OpenResFile?
resChanged      = 2     Resource changed?
```

Data Types

```
typedef unsigned long      FourCharCode;
typedef FourCharCode       ResType;
```

Functions

Reading Resources Into Memory

```
Handle    GetResource(ResType theType,short theID);
Handle    Get1Resource(ResType theType,short theID);
void      LoadResource(Handle theResource);
```

Disposing of Resources

```
void      ReleaseResource(Handle theResource);
```

Checking for Errors

```
short     ResError(void);
```

Demonstration Program

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// SysMemRes.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
//
// This program:
//
// •   Initialises the system software managers.
//
// •   Creates a nonrelocatable block of memory for a window structure.
//
// •   Loads a window template ('WIND') resource and creates a window.
//
// •   Loads a purgeable 'PICT' resource and a non-purgeable 'STR ' resource and draws
//     them in the window.
//
// •   Checks if any error codes were generated as a result of calls to to Memory Manager
//     and Resource Manager functions.
//
// •   Terminates when the mouse button is clicked.
//
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

//
..............................................................................................................................................................
........................................ includes

◊include <MacMemory.h>
◊include <Resources.h>
◊include <Sound.h>
◊include <TextUtils.h>
```

```
// ..........................................................................................................................................................................
........................................... defines

◊define rWindowResourceID     128
◊define rStringResourceID    128
◊define rPictureResourceID   128

// ..........................................................................................................................................................................
......... function prototypes

void  main                           (void);
void  doInitManagers         (void);
void  doNewWindow                 (void);
void  doDrawPictAndString    (void);

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ main

void  main(void)
{
    doInitManagers();
    doNewWindow();
    doDrawPictAndString();
    while(!Button()) ;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ initManagers

void  doInitManagers(void)
{
    MaxApplZone();
    MoreMasters();

    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NULL);

    InitCursor();
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doNewWindow

void  doNewWindow(void)
{
    WindowPtr windowPtr;
    Ptr                windowRecPtr;

    windowRecPtr = NewPtr(sizeof(WindowRecord));
    if(windowRecPtr == NULL)
    {
        SysBeep(10);
        ExitToShell();
    }

    windowPtr = GetNewCWindow(rWindowResourceID,windowRecPtr,(WindowPtr) -1);
    if(windowPtr == NULL)
    {
        SysBeep(10);
        ExitToShell();
    }

    SetPort(windowPtr);
    TextFont(systemFont);

}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doDrawPictAndString

void  doDrawPictAndString(void)
{
    PicHandle         pictureHdl;
    StringHandle    stringHdl;
    Rect                  pictureRect;
    SInt16              resourceError;
    OSErr                   memoryError;
```

```
    pictureHdl = GetPicture(rPictureResourceID);
    if(pictureHdl == NULL)
    {
        SysBeep(10);
        ExitToShell();
    }

    SetRect(&pictureRect,148,25,353,170);

    HNoPurge((Handle) pictureHdl);
    DrawPicture(pictureHdl,&pictureRect);
    HPurge((Handle) pictureHdl);

    stringHdl = GetString(rStringResourceID);
    if(stringHdl == NULL)
    {
        SysBeep(10);
        ExitToShell();
    }

    MoveTo(105,210);
    DrawString(*stringHdl);

    ReleaseResource((Handle) pictureHdl);
    ReleaseResource((Handle) stringHdl);

    resourceError = ResError();
    if(resourceError == noErr)
    {
        MoveTo(162,240);
        DrawString("\pNo Resource Manager errors");
    }

    memoryError = MemError();
    if(memoryError == noErr)
    {
        MoveTo(165,255);
        DrawString("\pNo Memory Manager errors");
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
```

Demonstration Program Comments

◊*includes*

The following explains the inclusion of each of the specified header files:

MacMemory.h **Prototypes:** MaxApplZone NewPtr HNoPurge HPurge MemError

Resources.h **Prototypes:** ReleaseResource ResError

Sound.h **Prototypes:** SysBeep

Sound.h itself includes Dialogs.h, which contains:
Prototypes: InitDialogs

Dialogs.h also includes TextEdit.h, which contains
Prototypes: TEInit

Dialogs.h also includes MacWindows.h, which contains:
Prototypes: InitWindows GetNewCWindow
Data Types: WindowRecord

MacWindows.h itself includes Events.h, which contains:
Prototypes: Button

Dialogs.h also includes Controls.h, which includes Menus.h, which contains:
Prototypes: InitMenus

Menus.h itself includes MacTypes.h, which contains:
DataTypes: SInt16, Ptr StringHandle Rect OSErr Handle
Constants: noErr

Menus.h also includes Fonts.h, which contains:
Prototypes: InitFonts
Constants: systemFont

Menus.h also includes Quickdraw.h, which contains:
Prototypes: InitGraf InitCursor SetPort SetRect DrawPicture MoveTo GetPicture
Data Types: WindowPtr PicHandle
QuickDraw Global Variable: thePort

Quickdraw.h itself includes QuickdrawText.h, which contains:
Prototypes: TextFont DrawString

Menus.h also includes Processes.h, which contains:
Prototypes: ExitToShell

TextUtils.h **Prototypes:** GetString

◊defines

Constants are established for the resource IDs of the 'WIND', 'PICT' and 'STR ' resources.

main

The main function calls the application-defined functions which initialise the system software managers, create a window, and draw a picture and text in the window. It then waits for a button click before terminating the program.

doInitManagers

doInitManagers grows the application heap, creates a block of master pointers, initialises the system software managers, and sets the cursor to the standard arrow shape.

Note that the function name is somewhat of a misnomer in that it does more than initialise the system software managers. However, since growing the heap, creating additional master pointer blocks, and setting the standard arrow cursor shape are invariably part of an application's setting up process, it is convenient to attend to those matters within the doInitManagers function. This practice will continue in all other demonstration programs.

The call to MaxApplZone is really not required for this simple program. However, it should be the first call in any serious application. The call grows the heap immediately to the maximum permissible size, assisting in the prevention of heap fragmentation, reducing the number of blocks which the Memory Manager has to purge when satisfying a memory request and speeding up memory allocation operations.

The call to MoreMasters to allocate a block of master pointers is really not required in this simple program because the Operating System automatically allocates one block of master pointers at application launch. However, in larger applications where more than 64 master pointers are required, the call, or calls, to MoreMasters should be made here so that all master pointer (nonrelocatable) blocks are located at the bottom of the heap. This will assist in preventing heap fragmentation.

The next six lines initialise certain system software managers. Not all of the data structures and variables inititialised by these calls will be used by this simple program; however, any serious application will require the full initialisation shown. It is also relevant that some managers require the use of information in other managers, so those other managers need to be initialised at least for that purpose. Some explanatory notes on the various calls are as follows:

- InitGraf initialises the QuickDraw global variables. The first element in the QuickDraw global data area is a pointer (thePort) to the current graphics port. Because it is the first QuickDraw global, passing its address to InitGraf tells QuickDraw where all the other QuickDraw globals are located. Other QuickDraw globals initialised by InitGraf are:
  - The pattern variables qd.white, qd.black, qd.gray, qd.ltGray, qd.dkGray.
  - qd.arrow, which contains the standard cursor arrow shape and which can be passed as an argument to QuickDraw's cursor functions.
  - qd.screenBits, a data structure which describes the main screen. The field screenBits.bounds contains a rectangle which encloses the main screen.
  - qd.randSeed, which is used to seed the random number generator.

  Note: The header file Quickdraw.h defines the following data type:

```
struct QDGlobals
{
   char        privates[76];
   long        randSeed;
   BitMap      screenBits;
   Cursor      arrow;
   Pattern     dkGray;
   Pattern     ltGray;
```

```
    Pattern     gray;
    Pattern     black;
    Pattern     white;
    GrafPtr     thePort;
};
typedef struct QDGlobals QDGlobals;
extern QDGlobals qd;
```

In the 680x0 environment, the runtime libraries define the QuickDraw global variable qd. There is no need for your application to do this.

- InitFont initialises the Font Manager and loads the system font into memory. Since the Window Manager uses the Font Manager to draw the window's title, etc., InitFonts must be called *before* InitWindows. Also, it must be called *after* InitGraf.

- InitWindows initialises the Window Manager port. It must be called *after* InitGraf and InitFonts. It draws the familiar rounded rectangle desktop with an empty menu bar at the top. The fill pattern used is the resource whose resource ID is represented by the constant deskPatID. (If a different fill pattern is required, it can be specified in the application's resource file.) The call establishes a nonrelocatable block (the Window Manager port) in the application heap.

- InitMenus allocates heap storage for the menu list and draws an empty menu bar. (For some unknown reason, InitWindows and InitMenus both draw the menu bar.) InitMenus must be called *after* InitGraf, InitFonts and InitWindows.

- TEInit initialises TextEdit, the Text editing manager, by allocating an internal handle for the TextEdit scrap (not the same as the "desk scrap" maintained by the Desk Manager). It should be called even if the application does not explicitly use TextEdit functions, since it ensures that dialog boxes and alert boxes work correctly.

- InitDialogs initialises the Dialog Manager and optionally installs a function to get control after a fatal system error. It installs the standard sound procedure (for alerts) and sets all text replacement parameters to empty strings (see the function ParamText).

InitCursor sets the cursor shape to the standard arrow cursor and sets the cursor level to 0, making it visible. (The 68-byte Cursor structure for the standard arrow cursor can be found in the QuickDraw data area.)

doNewWindow

doNewWindow creates a window.

NewPtr is used to allocate a nonrelocatable block of memory for the window structure. If the call is not successful, the system alert sound is played and the program is terminated by a call to ExitToShell, which releases the heap and hands control to the Finder. (Note that error handling here, and in the rest of the program, is thus somewhat rudimentary. Note also that SysBeep's parameter is nowadays ignored, but must be included for historical reasons.)

The call to GetNewCWindow creates a window using the 'WIND' template resource specified in the first parameter, and using the pointer to the nonrelocatable block already allocated for the window structure as the second parameter. (The third parameter tells the Window Manager to open the window in front of all other windows.) The type, size, location, appearance, title and visibility of the window are all established by the 'WIND' resource.

Recall that, as soon as the data from the 'WIND' template resource is copied to the window structure during the creation of the window, the nonrelocatable block occupied by the template will automatically be marked as purgeable.

The call to SetPort makes the new window's graphics port the current port for drawing operations. The call to TextFont at sets the font for that port to the standard system default font (Chicago or Charcoal, depending on the setting in the Appearance control panel).

doDrawPictAndString

doDrawPictAndString draws a picture and some text strings in the window.

GetPicture reads in the 'PICT' resource corresponding to the ID specified in the GetPicture call. If the call is not successful, the system alert sound is played and the program terminates.

The SetRect call assigns values to the left, top, right and bottom fields of a Rect variable. This Rect is required for a later call to DrawPicture.

The basic rules applying to the use of purgeable resources are to load it, immediately make it unpurgeable, use it immediately, and immediately make it purgeable. Accordingly, the HNoPurge call makes the relocatable block occupied by the resource unpurgeable, the DrawPicture call draws the picture in the window's graphics port, and the HPurge call makes the relocatable block purgeable again.

Note that, because HNoPurge and HPurge expect a parameter of type Handle, pictureHdl (a variable of type PicHandle) must be cast to a variable of type Handle.

GetString then reads in the specified 'STR ' resource. Once again, if the call is not successful, the system alert sound is played and the program terminates. MoveTo moves the graphics "pen" to an appropriate position before DrawString draws

the string in the window's graphics port. (Since the 'STR ' resource, unlike the 'PICT' resource, does not have the purgeable bit set, there is no requirement to take the precaution of a call to HNoPurge in this case.)

Note the parameter in the call to DrawString. stringHdl, like any handle, is a pointer to a pointer. It contains the address of a master pointer which, in turn, contains the address of the data. Dereferencing the handle once, therefore, get the required parameter for DrawString, which is a pointer to a string.

The calls to ReleaseResource release the 'PICT' and 'STR ' resources. These calls release the memory occupied by the resources and set the associated handles in the resource map in memory to NULL.

The ResError call returns the error code of the most recent resource-related operation. If the call returns noErr (indicating that no error occurred as a result of the most recent call by a Resource Manager function), some advisory text is drawn in the graphics port.

The next six lines examine the result of the most recent call to a memory manager function and draw some advisory text if no error occurred as a result of that call.

Note that the last two calls to DrawString utilise "hard-coded" strings. This sort of thing is discouraged in the Macintosh programming environment. Such strings should ordinarily be stored in a 'STR◊' (string list) resource rather than hard-coded into the source code. The \p token causes the compiler to compile these strings as **Pascal strings**.

PASCAL STRINGS

As stated in the Preface, when it comes to the system software, the ghost of the Pascal language forever haunts the C programmer. For example, a great many system software functions take Pascal strings as a required parameter, and some functions return Pascal strings.

Pascal and C strings differ in their formats. A C string comprises the characters followed by a terminating 0 (or NULL byte):

In a Pascal string, the first byte contains the length of the string, and the characters follow that byte:

Not surprisingly, then, Pascal strings are often referred to as "length-prefixed" strings.

In Chapter 3, you will encounter the data type Str255. Str255 is the C name for a Pascal-style string capable of holding up to 255 characters. As you would expect, the first byte of a Str255 holds the length of the string and the following bytes hold the characters of the string.

Utilizing 256 bytes for a string will simply waste memory in many cases. Accordingly, the header file Types.h defines the additional types Str63, Str32, Str31, Str27, and Str15, as well as the Str255 type:-

```
typedef unsigned char Str255[256];
typedef unsigned char Str63[64];
typedef unsigned char Str32[33];
typedef unsigned char Str31[32];
typedef unsigned char Str27[28];
typedef unsigned char Str15[16];
```

Note, then, that a variable of type Str255 holds the address of an array of 256 elements, each element being one byte long.

As an aside, in some cases you may want to use C strings, and use standard C library functions such as strlen, strcpy, etc., to manipulate them. Accordingly, be aware that functions exist (C2PStr, P2CStr) to convert a string from one format to the other.

You may wish to make a "working" copy of the SysMemRes demonstration program file package and, using the working copy of the source code file SysMemRes.c, replace the function doDrawPictAndString with the following, compile-link-run, and note the way the second and third strings appear in the window.

```
void  doDrawPictAndString(void)
{
  Str255  string1 = "\pIs this a Pascal string I see before me?";
  Str255  string2 = "Is this a Pascal string I see before me?";
  Str255  string3 = "%s this a Pascal string I see before me?";
  Str255  string4;
  SInt16  a;

  // Draw string1
```

```
  MoveTo(30,100);
  DrawString(string1);

  // Change the length byte of string1 and redraw

  string1[0] = (char) 23;
  MoveTo(30,120);
  DrawString(string1);

  // Leave the \p token out at your peril
  // I (ASCII code 73) is now interpreted as the length byte

  MoveTo(30,140);
  DrawString(string2);

  // More peril:-  % (ASCII code 37) is now the length byte

  MoveTo(30,160);
  DrawString(string3);

  // A hand-built Pascal string

  for(a=1;a<27;a++)
    string4[a] = (char) a + 64;

  string4[0] = (char) 26;

  MoveTo(30,180);
  DrawString(string4);

  // But is there a Mac OS function to draw the C strings correctly?

  MoveTo(30,200);
  DrawText(string2,0,40);  // Look it up in your on-line reference
}
```