1

SYSTEM SOFTWARE, MEMORY, AND RESOURCES
Includes Demonstration Program SysMemResPascal

Macintosh System Software

All Macintosh applications make many calls, for many purposes, to Macintosh **system software** routines. 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 routines are components of either the **Macintosh Toolbox** or the **Macintosh Operating System**. In essence:
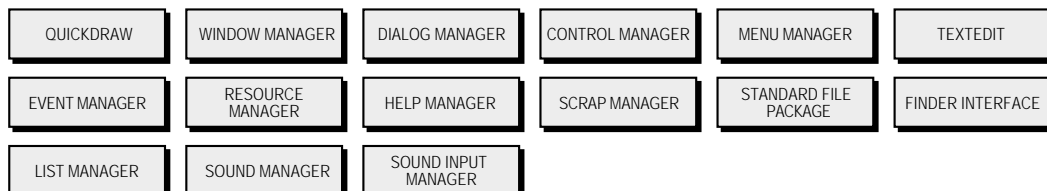
•	Toolbox routines have to do with mediating your application with the user. They relate, in general, to the management of elements of the user interface.

•	Operating System routines 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 routines is further divided into functional groups which are usually known as **managers.**[2]

Toolbox

The Toolbox comprises the following managers:

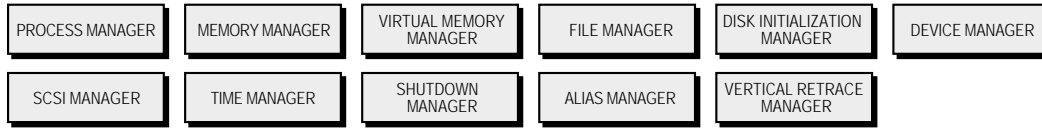|					
QUICKDRAW	WINDOW MANAGER	DIALOG MANAGER	CONTROL MANAGER	MENU MANAGER	TEXTEDIT
EVENT MANAGER	RESOURCE MANAGER	HELP MANAGER	SCRAP MANAGER	STANDARD FILE PACKAGE	FINDER INTERFACE
LIST MANAGER	SOUND MANAGER	SOUND INPUT MANAGER			

---

[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 routines are referred to as **packages.** In general, the distinction between managers and packages is unimportant. Packages are nowadays generally referred to as managers.

Operating System

The Operating System comprises the following managers:

| PROCESS MANAGER	MEMORY MANAGER	VIRTUAL MEMORY MANAGER	FILE MANAGER	DISK INITIALIZATION MANAGER	DEVICE MANAGER
SCSI MANAGER	TIME MANAGER	SHUTDOWN MANAGER	ALIAS MANAGER	VERTICAL RETRACE MANAGER	

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) are as follows:

FONT MANAGER	TEXT UTILITIES	SCRIPT MANAGER	TEXT SERVICES MANAGER

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

APPLE EVENT MANAGER	EVENT MANAGER	PROGRAM-TO-PROGRAM COMMUNICATIONS TOOLBOX
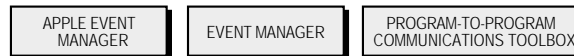
- **QuickTime.** QuickTime is a collection of managers and other system software components which allow an application to control time-based data such as video clips, animation sequences and sound sequences. It comprises:

MOVIE TOOLBOX	IMAGE COMPRESSION MANAGER	A SET OF PREDEFINED COMPONENTS

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

CONNECTION MANAGER	TERMINAL MANAGER	FILE TRANSFER MANAGER	COMMUNICATIONS RESOURCE MANAGER

System Software Routines

Routines in ROM

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

Routines 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**.

Routines in RAM — Extensions

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

Glue Routines

Some routines declared in a particular development system's header files are provided by the development system itself, not by the system software. These routines are known as **glue routines** and are constructed by modifying available system software routines in some way. Knowing whether a particular routine 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. Fig 1 illustrates the arrangement of memory when several applications are open at once.
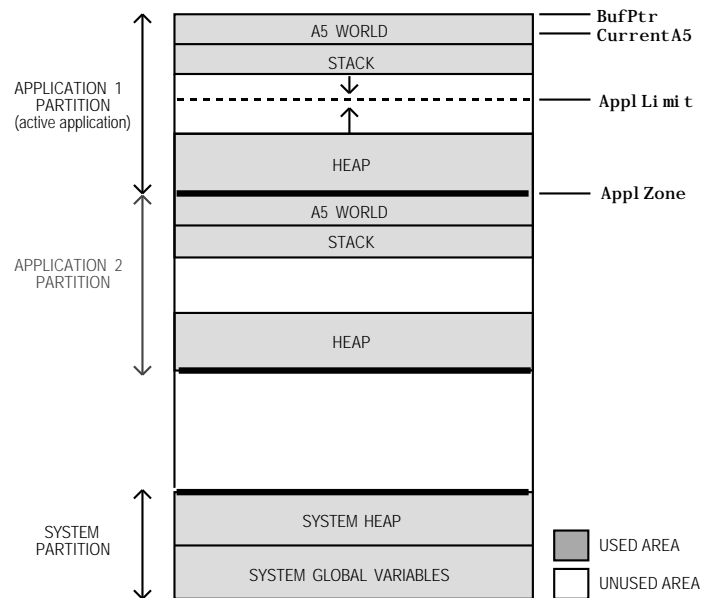

FIG 1 - MEMORY ORGANIZATION IN THE COOPERATIVE MULTITASKING ENVIRONMENT

---

[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.)

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.[4]

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).

• 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 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 routine, space is automatically allocated on the stack for a **stack frame**, which contains the routine's parameters, local variables and return address. Once the call is executed, the local variables and function parameters are popped off the stack.[5]

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.[6]

By default, the stack can grow to 8KB on Macintoshes without Color QuickDraw and 24KB on Macintoshes with Color QuickDraw. 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.[7]

---

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

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

[6] 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.

[7] 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.

If necessary, you can change the default size of the stack using the routine `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) which are currently loaded into memory.

- Other dynamically allocated items such as window records, dialog records, 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 routine which itself calls a Memory Manager memory allocation routine.

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 routines 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 routines 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),
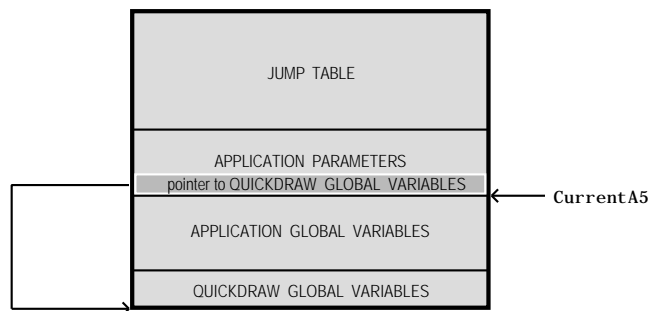

FIG 2 - THE A5 WORLD

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.

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 in the Universal Interface file `types.p` as follows:

```
type
Ptr = ^SignedByte; { 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 routine `NewPtr` allocates a nonrelocatable block, for example:

```
var
myPointer : Ptr;

...
myPointer := NewPtr(sizeof(WindowRecord));
```

Nonrelocatable blocks are disposed of by a call to `DisposePtr`.[8]

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.[9]

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 3. `Handle` is defined as follows:

```
type
Handle = ^Ptr;   {A pointer to a master pointer.}
```

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

```
var
myHandle : Handle;

...
myHandle := NewHandle(sizeof(myDataRecord));
```

---

[8]Many system software routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by the development environment. For example, the previous name for this routine was `DisposPtr`.

[9]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.)

| Relocatable block | Nonrelocatable block |

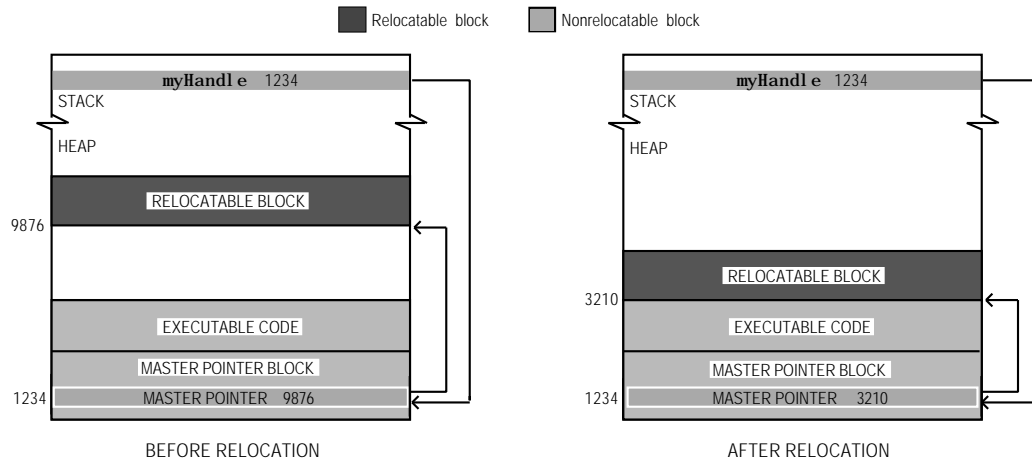**BEFORE RELOCATION**

AFTER RELOCATION

FIG 3 - A HANDLE TO A RELOCATABLE BLOCK

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 `NIL` to the handle variable after you dispose of the 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[10], use that pointer within a loop to read or write data to or from the block. If, within that loop, you call a routine which has the potential to move memory, and if that routine 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 4.

---

[10]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.
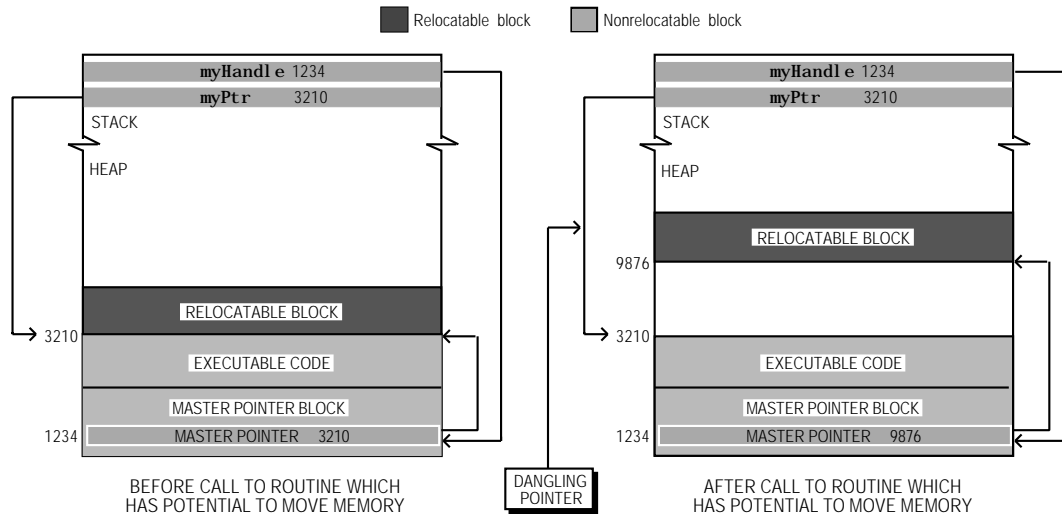
Relocatable block	Nonrelocatable block

FIG 4 - A DANGLING POINTER

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

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

```
type

  MyRecord = record
    intArray : array [1..1000] of integer;
    ch : char;
    end;
  MyRecordPtr = ^MyRecord;
  MyRecordHandle = ^MyRecordPtr;

...

procedure MyProcedure;

var
  theHdl : MyRecordHandle;
  thePtr : MyRecordPtr;
  count : integer;

begin

  theHdl := MyRecordHandle(NewHandle(sizeof(MyRecord)));

  HLock(theHdl);                    {Lock the relocatable block ...}
  thePtr := theHdl^;                {because the handle has been dereferenced ...}

  for count := 1 to 1000 do
  begin
    thePtr^.intArray[count] := 0; {and used in this loop ...}
    DrawChar(char('A'));          {which calls a routine which could cause}
  end;                            {the relocatable block to be moved.}

  HUnlock(theHdl);                  {On loop exit, unlock the relocatable block.}
end;

{DrawChar is defined in QuickDrawText.p as

  procedure DrawChar(ch: char);   }
```

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 routines which could move or purge memory.

If the handle's master pointer is set to `NIL`, 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**[11]. 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

---

[11]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.

fragmentation if relocatable blocks have previously been moved to the previous top of the heap and locked.)[12]

- 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 routines `HGetState` and `HSetState` come in. The sequence of events in Function C should have been as follows:

```
theTagByte : SInt8;
...
theTagByte := HGetState(myHandle); { Whatever the current state is, save it.}
HLock(myHandle);                    { Redundant if Procedure 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 }
                                    { been locked.  It could have been unlocked.)}
end;
```

---

[12]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.

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.[13]

Temporary Memory

Each application is limited to a particular memory partition, whose size is determined by information in the application's `'SIZE'` **resource** (see Chapter 2 — Low-Level and Operating System Events). The size of the partition places certain limitations on the size of the application heap and thus on the size of the buffers and other data structures used by the application. In general, the partition's size should be large enough to hold all the buffers, resources and other data likely to be needed by the application during execution.

However, if for some reason more memory is required than is currently available in the application heap, the Operating System can be requested to make available any memory that is not yet allocated to any other application. This memory, known as **temporary memory**, is usually not contiguous with memory in the application's zone.

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**.

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.

On suitably equipped Macintosh computers (that is, those with a 32-bit Memory Manager), the Operating System supports 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 routines to achieve the same results. You must avoid such practices if your application is to be 32-bit clean.

Memory Manager Errors

The low-memory address $0220, which is represented by the symbolic name `MemErr`, contains the error code resulting from the last call to a Memory Manager routine. 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.

---

[13]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.

- 109   nilHandleErr   Illegal operation on a NIL handle.

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 procedure:

```
procedure theProcedure;

   var
   thePointer : Ptr;
   osError : OSErr;

   begin
   thePointer := NewPtr(10000);
   if (MemError = memFullErr) then
     DoErrorAlert(eOutOfMemory);

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

   end;
```

When `theProcedure` returns, the 10000-byte nonrelocatable block will still exist (even though, incidentally, the 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 `theProcedure` 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.[14]

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:

---

[14]The dynamic memory inspection tool ZoneRanger, which is included with MetroWerks CodeWarrior, can be used to check your application for memory leaks.

- **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 code and, typically, resources which describe the application's windows, menus, controls, dialog boxes, icons, etc. Fig 5 illustrates the typical contents of the data and resource forks of an application file and a document file.

APPLICATION FILE   DOCUMENT FILE

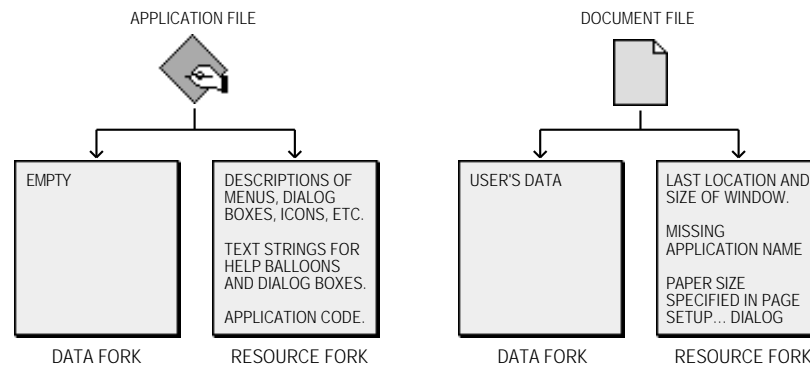| EMPTY	DESCRIPTIONS OF MENUS, DIALOG BOXES, ICONS, ETC. TEXT STRINGS FOR HELP BALLOONS AND DIALOG BOXES. APPLICATION CODE.	USER'S DATA	LAST LOCATION AND SIZE OF WINDOW. MISSING APPLICATION NAME PAPER SIZE SPECIFIED IN PAGE SETUP… DIALOG
DATA FORK	RESOURCE FORK	DATA FORK	RESOURCE FORK

FIG 5 - TYPICAL CONTENTS OF DATA FORKS AND RESOURCE FORKS IN APPLICATION AND DOCUMENT FILES

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 routines 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 15 — 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.[15]

- **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 procedures** (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 routines to create these elements, based on their resource descriptions, as needed.

You can create resource descriptions using a resource editor such as ResEdit (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.[16] 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.) */
  zoomDocProc,                /* Window definition ID. */
  invisible,                  /* Window is initially invisible. */
  goAway,                     /* Window has a close box. */
  $0,                         /* Reference constant. */
  "untitled",                 /* Window title. */
  staggerParentWindowScreen   /* Optional positioning specification. */
};
```

---

[15]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.

[16]Examples of resource descriptions in this and later chapters are in Rez input format, since this facilitates a clearer understanding of the elements which comprise the complete resource definition. Some chapters include instructions for creating specific resources using ResEdit. These instructions relate the ResEdit presentation and interface to the resource definitions in Rez input format as shown in the relevant chapter.

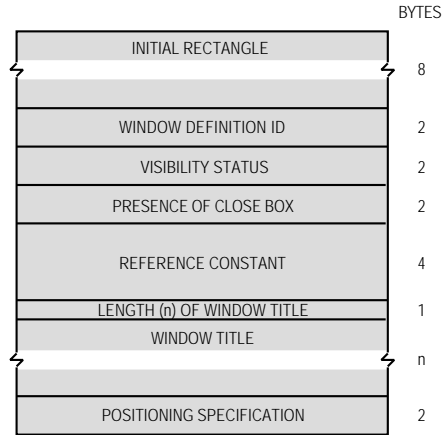The structure of the compiled `'WIND'` resource is shown at Fig 6.

BYTES

INITIAL RECTANGLE	8
WINDOW DEFINITION ID	2
VISIBILITY STATUS	2
PRESENCE OF CLOSE BOX	2
REFERENCE CONSTANT	4
LENGTH (n) OF WINDOW TITLE	1
WINDOW TITLE	n
POSITIONING SPECIFICATION	2

FIG 6 - STRUCTURE OF A COMPILED WINDOW (`'WIND '`) RESOURCE

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 procedure** (specified by the constant `zoomDocProc`) to be used to draw it. Definition procedures, 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 procedure specified by the constant `zoomDocProc` is contained within the `'WDEF'` resource with ID 0 in the System file. Note that it is possible to write your own custom window definition procedure (and, indeed, custom definition procedures 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		•		NIL
MENU	128	•			123C
WIND	128			•	NIL
PICT	128			•	NIL
PICT	129			•	NIL

Note that the handle entry in the resource map contains NIL 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 `GetNewWindow` is called to create a new window (specifying **128** as the resource ID), `GetNewWindow`, in turn, calls the Resource Manager function `GetResource`. `GetResource` loads the resource (assuming that it is not currently in memory), returns the handle to `GetNewWindow`, 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 functions directly, for example:

```
const
rPicture1 = 128;
rPicture2 = 129;
...
var
pic1Hdl : PicHandle;
pic2Hdl : PicHandle;
...
pic1Hdl := GetPicture(rPicture1);
pic2Hdl := GetPicture(rPicture2);

{GetPicture is defined in QuickDraw.p as

function GetPicture(pictureID: integer): PicHandle;  }
```

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 NIL 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 NIL, 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 <> nil) then               {If the handle to first resource is not NIL ...}
  DrawPicture(pic1Hdl,...);            {... draw the second picture.}
if (pic2Hdl <> nil) then               {If the handle to second resource is not NIL...}
  DrawPicture(pic2Hdl,...);            {... 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 NIL 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` procedure).

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 <> nil) then             {If the resource was successfully loaded ...}
  begin
  HNoPurge(Handle(pic1Hdl));         {make the resource unpurgeable ...}
  DrawPicture(pic1Hdl,...);          {draw the first picture ...}
  HPurge(Handle(pic1Hdl));           {and make the resource purgeable again.}
  end;

pic2Hdl := GetResource(rPicture2);   {Repeat for the second 'PICT' resource.}
if (pic2Hdl <> nil) then
  begin
  HNoPurge(Handle(pic2Hdl));
  DrawPicture(pic1Hdl,...);
  HPurge(Handle(pic2Hdl));
  end;
```

Note that this procedure only applies when you use routines 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 routine `GetNewWindow`) because routines 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 routine which gets resources directly, you should call the appropriate procedure 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 `NIL`.

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

•   `GetNewWindow` calls `GetResource` to read in the window resource template whose ID is specified in the `GetNewWindow` 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 record.

- Data is copied from the resource template into certain fields in the window record.

- The window template's block is then marked as purgeable.

Resource Manager Errors

The low-memory address $0A60, which is represented by the symbolic name ResErr, contains the error code resulting from the last call to a Resource Manager routine. 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 Routines

Data Types

```
SignedByte = -128..127;     {Arbitrary Byte in Memory}
Ptr = ^SignedByte;          {Pointer to nonrelocatable block.}
Handle = ^Ptr;              {Handle to relocatable block.}
Size = LONGINT;             {Size of a block in bytes.}
```

Routines

Setting Up the Application Heap

```
procedure   MaxApplZone;
procedure   MoreMasters;
function    GetApplLimit: Ptr;
procedure   SetApplLimit(zoneLimit: UNIV Ptr);
```

Allocating and Releasing NonRelocatable Blocks of Memory

```
function    NewPtr(byteCount: Size): Ptr;
function    NewPtrClear(byteCount: Size): Ptr;
function    NewPtrSys(byteCount: Size): Ptr;
function    NewPtrSysClear(byteCount: Size): Ptr;
procedure   DisposePtr(p: Ptr);
```

Allocating and Releasing Relocatable Blocks of Memory

```
function    NewHandle (byteCount: Size): Handle;
function    NewHandleClear(byteCount: Size): Handle;
function    NewHandleSys(byteCount: Size): Handle;
function    NewHandleSysClear(byteCount: Size): Handle;
function    NewEmptyHandle: Handle;
function    NewEmptyHandleSys: Handle;
procedure   DisposeHandle(h: Handle);
```

Changing the Size of Nonrelocatable and Relocatable Blocks

```
function    GetPtrSize(p: Ptr): Size;
procedure   SetPtrSize(p: Ptr; newSize: Size);
function    GetHandleSize(h: Handle): Size;
procedure   SetHandleSize(h: Handle; newSize: Size);
```

Getting and Setting the Properties of Relocatable Blocks

```
procedure   Hlock(h: Handle);
procedure   HUnlock(h: Handle);
procedure   HPurge(h: Handle);
procedure   HNoPurge(h: Handle);
function    HGetState(h: Handle): SInt8;
procedure   HSetState(h: Handle; flags: ByteParameter);
```

Managing Relocatable Blocks

```
procedure   EmptyHandle(h: Handle);
procedure   ReallocateHandle(h: Handle; byteCount: Size);
function    RecoverHandle(p: Ptr): Handle;
procedure   ReserveMem(cbNeeded: Size);
procedure   ReserveMemSys(cbNeeded: Size);
procedure   MoveHHi(h: Handle);
procedure   HLockHi(h: Handle);
```

Manipulating Blocks of Memory

```
procedure   BlockMove(srcPtr: UNIV Ptr; destPtr: UNIV Ptr; byteCount: Size);
procedure   BlockMoveData(srcPtr: UNIV Ptr; destPtr: UNIV Ptr; byteCount: Size);
function    PtrToHand(srcPtr: UNIV Ptr; VAR dstHndl: Handle; size: LONGINT): OSErr;
function    PtrToXHand(srcPtr: UNIV Ptr; dstHndl: Handle; size: LONGINT): OSErr;
function    HandToHand(var theHndl: Handle): OSErr;
function    HandAndHand(hand1: Handle; hand2: Handle): OSErr;
function    PtrAndHand(ptr1: UNIV Ptr; hand2: Handle; size: LONGINT): OSErr;
```

Accessing Memory Conditions

```
function    FreeMem: longint;
function    FreeMemSys: LONGINT;
function    MaxBlock: longint;
function    MaxBlock: LONGINT;
function    MaxBlockSys: LONGINT;
procedure   PurgeSpace(VAR total: LONGINT; VAR contig: LONGINT);
function    StackSpace: LONGINT;
```

Freeing Memory

```
function    CompactMem(cbNeeded: Size): Size;
function    CompactMemSys(cbNeeded: Size): Size;
procedure   PurgeMem(cbNeeded: Size);
procedure   PurgeMemSys(cbNeeded: Size);
function    MaxMem(var grow: Size): Size;
function    MaxMemSys(VAR grow: Size): Size;
```

Allocating Temporary Memory

```
function    TempNewHandle(logicalSize: Size; var resultCode: OSErr): Handle;
function    TempFreeMem: longint;
function    TempMaxMem(var grow: Size): Size;
```

Checking for Errors

```
function    MemError: OSErr;
```

Main Resource Manager Constants and Routines

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?}
```

Routines

Reading Resources Into Memory

```
function    GetResource(theType: ResType; theID: integer): Handle;
function    Get1Resource(theType: ResType; theID: integer): Handle;
procedure   LoadResource(theResource: Handle);
```

Disposing of Resources

```
procedure    ReleaseResource(theResource: Handle);
```

Checking for Errors

```
function    ResError: integer;
```

Demonstration Program

```
1    { ################################################################################
2    // SysMemResPascal.p
3    // ################################################################################
4    //
5    // This program:
6    //
7    // •   Initialises the system software managers.
8    //
9    // •   Creates a nonrelocatable block of memory for a window record.
10   //
11   // •   Loads a window template ('WIND') resource and creates a window.
12   //
13   // •   Loads a purgeable 'PICT' resource and a non-purgeable 'STR ' resource and draws
14   //     them in the window.
15   //
16   // •   Checks if any error codes were generated as a result to Memory Manager and
17   //     Resource Manager routines.
18   //
19   // •   Terminates when the mouse button is clicked.
20   //
21   // ################################################################################ }
22
23
24   program SysMemRes(input, output);
25
26   { .................................................................. include the following Universal Interfaces }
27
28   uses
29
30     Windows, Fonts, Menus, TextEdit, Dialogs, TextUtils, Resources, Memory,
31     QuickDraw, Types, Sound, Processes, OSUtils, QuickDrawText, Events, Segload;
32
33   { .................................................................. define the following constants }
34
35   const
36
37   rWindowResourceID = 128;
38   rStringResourceID = 128;
39   rPictureResourceID = 128;
40
41   { ######################################################## initialise various Managers }
42
43   procedure DoInitManagers;
44
45     begin
46     MaxApplZone;
47     MoreMasters;
48     InitGraf(@qd.thePort);
49     InitFonts;
50     InitWindows;
51     InitMenus;
52     TEInit;
53     InitDialogs(nil);
54     InitCursor;
55     end;
56       {of procedure DoInitManagers}
57
58   { ########################################################################## DoNewWindow }
59
60   procedure DoNewWindow;
61
62     var
63     myWindowPtr : WindowPtr;
```

```
64       windowRecPtr : Ptr;

65

66       begin
67       windowRecPtr := NewPtr(sizeof(WindowRecord));
68       if (windowRecPtr = nil) then
69          begin
70          SysBeep(10);
71          ExitToShell;
72          end;

73

74       myWindowPtr := GetNewWindow(rWindowResourceID, windowRecPtr, WindowPtr(-1));

75

76       if (myWindowPtr = nil) then
77          begin
78          SysBeep(10);
79          ExitToShell;
80          end;

81

82       SetPort(myWindowPtr);
83       TextFont(systemFont);

84

85       end;
86          {of procedure DoNewWindow}

87

88    { ################################################################### DoDrawPictAndString }

89

90    procedure DoDrawPictAndString;

91

92       var
93       pictureHdl : PicHandle;
94       stringHdl : StringHandle;
95       pictureRect : Rect;
96       resourceError : integer;
97       memoryError : OSErr;

98

99       begin
100      pictureHdl := GetPicture(rPictureResourceID);
101      if (pictureHdl = nil) then
102         begin
103         SysBeep(10);
104         ExitToShell;
105         end;

106

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

108

109      HNoPurge(Handle(pictureHdl));
110      DrawPicture(pictureHdl, pictureRect);
111      HPurge(Handle(pictureHdl));

112

113      stringHdl := GetString(rStringResourceID);
114      if (stringHdl = nil) then
115         begin
116         SysBeep(10);
117         ExitToShell;
118         end;

119

120      MoveTo(105, 210);
121      DrawString(stringHdl^^);

122

123      ReleaseResource(Handle(pictureHdl));
124      ReleaseResource(Handle(stringHdl));

125

126      resourceError := ResError;
127      if (resourceError = noErr) then
128         begin
129         MoveTo(162, 240);
130         DrawString('No Resource Manager errors');
131         end;

132

133      memoryError := MemError;
134      if (memoryError = noErr) then
135         begin
136         MoveTo(165, 255);
137         DrawString('No Memory Manager errors');
138         end;

139
```

```
140      end;
141        {of procedure DoDrawPictAndString}
142
143   { ############################################################ start of main program }
144
145   begin
146
147   DoInitManagers;
148   DoNewWindow;
149   DoDrawPictAndString;
150   while not Button do {nothing};
151
152   end.
153
154   { ########################################################################### }
```

Demonstration Program Comments

The uses block

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

Windows.p	Routines:	InitWindows GetNewWindow
	Data Types:	WindowRecord
Memory.p	Routines:	MaxApplZone NewPtr HNoPurge HPurge MemError
Types.p	Data Types:	Ptr StringHandle Rect OSErr Handle
	Constants:	noErr
Quickdraw.p	Routines:	InitGraf InitCursor SetPort SetRect DrawPicture MoveTo GetPicture
	Data Types:	WindowPtr PicHandle
	QuickDraw Global Variable:	thePort
QuickdrawText.p	Routines:	TextFont DrawString
Events.p	Routines:	Button
OSUtils.p	Routines:	SysBeep
Fonts.p	Routines:	InitFonts
	Constants:	systemFont
Menus.p	Routines:	InitMenus
TextEdit.p	Routines:	TEInit
Dialogs.p	Routines:	InitDialogs
Processes.p	Routines:	ExitToShell
TextUtils.p	Routines:	GetString
Resources.p	Routines:	ReleaseResource ResError

The constant declaration block

The constant declaration block establishes constants for the resource IDs of the 'WIND', 'PICT' and 'STR ' resources.

The procedure 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 procedure 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 following 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. This is the reason for use of the @ operator - to ensure it's the address of thePort that gets passed to InitGraf. 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 routines.

  - 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 interface file Quickdraw.p defines the following data type:

  ```
  type
  QDGlobals = record
    privates:      packed array [0..75] OF char;
    randSeed:      longint;
    screenBits:    BitMap;
    arrow:         Cursor;
    dkGray:        Pattern;
    ltGray:        Pattern;
    gray:          Pattern;
    black:         Pattern;
    white:         Pattern;
    thePort:       GrafPtr;
  end;

  var
  qd: QDGlobals;
  ```

  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 routine 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).

At the final line of this procedure, 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.)

The procedure DoNewWindow

DoNewWindow creates a window.

Firstly NewPtr is used to allocate a nonrelocatable block of memory for the window record. If the call is not successful (windowRecPtr is NIL), the system alert sound is played (SysBeep) 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 GetNewWindow 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 record 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 record 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 sets the font for that port to the standard system default font (Chicago).

The procedure DoDrawPictAndString

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

Line 100 reads in the 'PICT' resource corresponding to the ID specified in the GetPicture call. If the call is not successful (Line 101), the system alert sound is played and the program terminates.

Line 107 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, Line 109 makes the relocatable block occupied by the unpurgeable, Line 110 draws the picture in the window's graphics port, and Line 111 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.

At Line 113, GetString reads in the specified 'STR ' resource. Once again, if the call is not successful, the system alert sound is played and the program terminates (Lines 114-118). Line 120 moves the graphics "pen" to an appropriate position before Line 121 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.)

Lines 123-124 release the 'PICT' and 'STR ' resources. The calls to ReleaseResource release the memory occupied by the resources and set the associated handles in the resource map in memory to NIL.

Line 126 calls ResError to return 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 routine), some advisory text is drawn in the graphics port (Lines 127-131).

Lines 133-138 examine the result of the most recent call to a memory manager routine and draw some advisory text if no error occurred as a result of that call.

Note that the calls to DrawString at Lines 130 and 137 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 main program block

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 (Lines 147-149). It then waits for a button click before terminating the program (Line 150).