1

SYSTEM SOFTWARE, POWERPC RUN-TIME ENVIRONMENT, MANAGING MEMORY, AND RESOURCES

Demonstration Program: SysMemRes

System Software

All Macintosh applications make many calls, for many purposes, to **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 entire collection of system software functions is divided into functional groups known as **managers**. In Carbon, each manager is part of a higher-level grouping called a **manager group**. The manager groups and their constituent managers are shown in the following, in which those managers of particular relevance to this book appear in bold typeface:

Application Utilities *Facilities for extending human interface features and data-sharing capabilities beyond those provided by the Human Interface Toolbox.*			
AltiVec Utilities	Find By Content	Translation Manager	
Carbon Core *Essential services that are generally devoid of a user interface, including memory and process management.*			
Alias Manager	**Collection Manager**	Component Manager	**File Manager**
Folder Manager	**Low Memory Accessors**	Memory Management Utilities	**Memory Manager**
Mixed Mode Manager	Multiprocessing Services	**Notification Manager**	Pascal String Utilities
Resource Manager	Thread Manager	Time Manager	
Common OS Services			
Display Manager	**Gestalt Manager**	Power Manager	**Process Manager**
Speech Recognition Manager	**Speech Synthesis Manager**		
Core Foundation *Abstracts several operating system utitlities fundamental to both Mac OS 8/9 and Mac OS X.*			
Base Services	Bundle Services	Collection Services	Plug-in Services
Preferences Services	**Property List Services**	**String Services**	URL Services
Utility Services	XML Services		
Graphics Services *Allows applications to construct and render images.*			
Carbon Printing Manager	**Color Picker Manager**	ColorSync Manager	HTML Rendering Library
Palette Manager	**QuickDraw Manager**		

[1] The main *other* open application that an application needs to work with is the **Finder**. 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.

Hardware Interfaces			
Device Manager	PCI Card Services		
Human Interface Toolbox *Comprises groups of libraries that implement the Mac OS human interface.*			
Appearance Manager	**Apple Help**	**Carbon Event Manager**	**Carbon Help Manager**
Control Manager	**Dialog Manager**	**Drag Manager**	**Event Manager**
Finder Interface	**Icon Utilities**	**List Manager**	**Menu Manager**
Navigation Services	**Picture Utilities**	**Scrap Manager**	**Window Manager**
Networking and Communication Services			
Internet Config	NSL manager	Open Transport	URL Access Manager
QuickTime *Assist your application in combining multiple forms of communication, such as text, pictures, video, and music.*			
Image Compression Manager	Media Handlers	Movie Toolbox	QuickTime Components
QuickTime Media Layer	QuickTime Music	QuickTime Streaming	QuickTime VR
Sound Manager			
Runtime Services *Provides programming interfaces that prepare code for execution and control how functions call each other.*			
Code Fragment Manager	Debugger Services	Math and Logical Utilities	
Scripting and Apple Events			
Apple Event Manager	Open Scripting Architecture		
Text and Other International Services *Assistance in creating applications for world-wide markets*			
Apple Type Services for Unicode Imaging	**Date, Time, and Measurement Utilities**	**Font Manager**	
FontSync	**Multilingual Text Engine**	**QuickDraw Text**	
Script Manager	Text Encoding Conversion Manager	Text Services Manager	
Text Utilities	**TextEdit**	Unicode Utilities	

PowerPC Run-Time Environment

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. The system software and your development system jointly determine the run-time environment.

Fragments

The PowerPC run-time model is based on the use of **fragments** as the standard way of organising executable code and data in memory. A fragment is any block of executable PowerPC code and its associated data. Fragments can be of any size, and are complete, executable entities.

There are three broad categories of fragments, namely, **applications**, **import libraries**, and **extension**s. (Import libraries and system extensions are sometimes called **shared libraries** or **dynamically-linked libraries**.) An application is a fragment that can be launched by the user from the Finder.

Code Fragment Resource

You use a code fragment (`'cfrg'`) resource (see Resources, below) to specify some characteristics of a code fragment. For an application, the code fragment resource indicates to the Process Manager that the application's data fork contains an executable code fragment container.

You do not have to create the `'cfrg'` resource yourself because CodeWarrior does this for you when you compile your application.

Organisation of Memory

The basic organisation of memory in the PowerPC run-time environment is as follows:

- A **system partition**, which is reserved for the exclusive use of the system, occupies the lowest memory address. Part of the system partition is occupied by the system global variables, most of which 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.
- Most of the remaining space is allocated to the Process Manager, which creates an **application partition** for each open application. Application partitions are loaded into the top part of RAM first.

The Application Partition

In each application partition, there is a **stack** and a **heap**, as well as space for the application's **global variables** (see Fig 1).

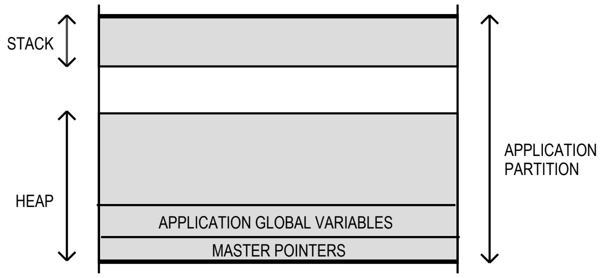

FIG 1 - STRUCTURE OF APPLICATION PARTITION

The Stack

The stack is used for memory allocation associated with the execution of functions. When an application calls a function, space is allocated on the stack for a **stack frame**, which contains the function's parameters, local variables and return address. The local variables and function parameters are popped off the stack when the function has executed,. The C compiler generates the code that creates and deletes stack frames for each function call.

It is important to understand that the Memory Manager has no way of preventing the stack from encroaching on, and corrupting, the heap. Ordinarily, 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.[2]

The Heap

The application heap is that part of the application partition in which memory is dynamically allocated and released on demand. 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.

Managing Memory — Mac OS 8/9

Inside the Application 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**.

[2]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.

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 a byte in memory. Thus a pointer to a nonrelocatable block is simply the address of the first byte in that block of memory. 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(myDataStructure));
```

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

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 relocatable 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 as to ensure that the master pointer always contains the current address of the relocatable block.

One master pointer block, which contains 64 master pointers, is allocated for your application by the Memory Manager at application launch. This block is located at the very bottom of the application heap (see Fig 1). The function `MoreMasterPointers` may be called by your application to allocate additional master pointers. To ensure that these additional (nonrelocatable) blocks are allocated as low in the heap as possible, the call to `MoreMasterPointers` should be made at the beginning of your program.[3]

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 2. `Handle` is defined as follows:

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

[3] 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. Your call to `MoreMasterPointers` should thus allocate sufficient master pointers to ensure that the Memory Manager never needs to create additional master pointer blocks for you. (You can empirically determine how many master pointers to allocate using a low-level debugger.)

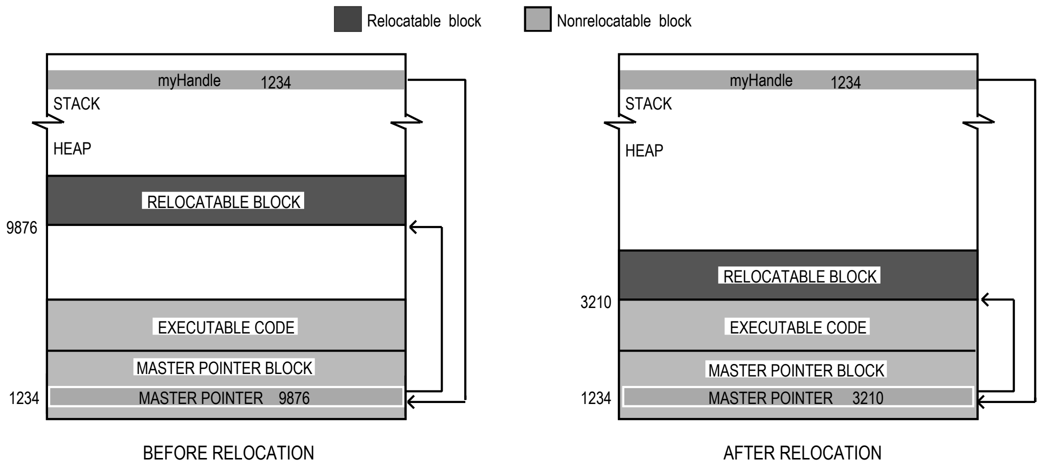

FIG 2 - A HANDLE TO A RELOCATABLE BLOCK

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

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

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 inadvertently use a handle to a block you have already disposed of, your application could crash. You can avoid these problems by assigning the value `NULL` 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[4], use that pointer within a loop to read or write data to or from the block. If, within that loop, you call a function that has the potential to move memory, and if that function actually causes the relocatable block to be moved, the master pointer will be correctly

[4] 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.

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 3.

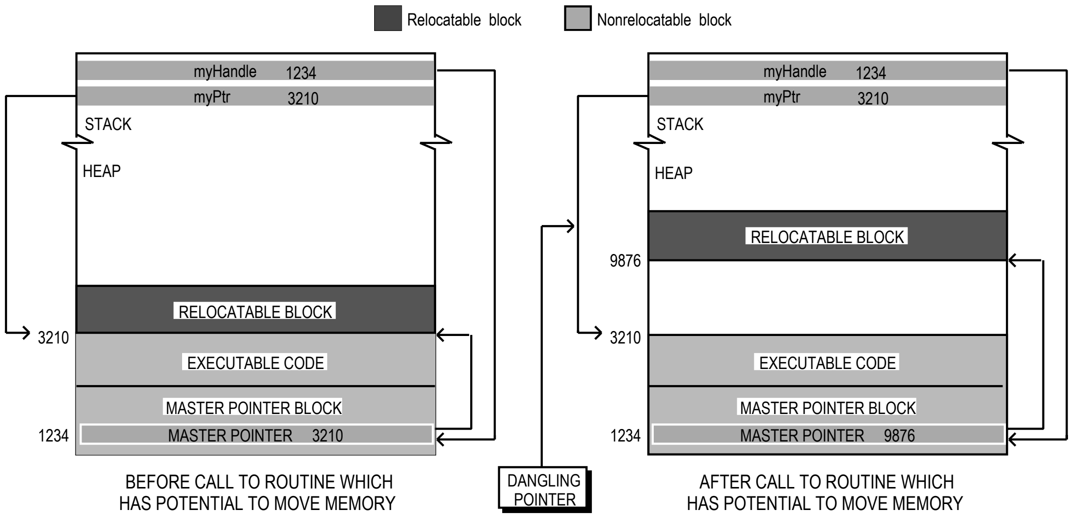

FIG 3 - A DANGLING POINTER

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.

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 that 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**[5]. 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 `MoreMasterPointers` to allocate at least as many (nonrelocatable) master pointers as are required during program execution.

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

· Avoid disposing of, and subsequently reallocating, nonrelocatable blocks during the application's execution.

· If you need to allocate a relocatable block that will need to be locked for a long period of time, call `ReserveMem` at the beginning of the program to reserve memory for the block as close to the bottom of the heap as is possible, and then lock the block immediately after allocating it.

[5] 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.

· 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 relocatable block to the top of the heap and then lock it. Unlock the block when it no longer needs to be locked.

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
                                   // 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.[6]

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:

[6] 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.

```
void  theFunction(void)
{
  Ptr   thePointer;
  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.[7]

Memory Manager Errors

The error code resulting from the last call to a Memory Manager function.may be retrieved by calling the function `MemError`. Some of the error codes which may be returned by `MemError` are as follows:

Value	*Constant*	*Description*
0	noErr	No error occurred.
-108	memFullErr	No room in heap.
-109	nilHandleErr	Illegal operation on a `NULL` handle.
-117	memLockedErr	Trying to move a locked block (`MoveHHi`).

Managing Memory — Mac OS X Considerations

Preamble — Memory in Mac OS X

In Mac OS X, each application runs in its own address space, meaning that an application cannot reference memory locations outside its assigned address space.

Compared with Mac OS 8/9, Mac OS X uses an entirely different heap structure and allocation behaviour.

Mac OS X uses a dynamic and highly efficient virtual memory system, which is always enabled. Carbon application must therefore assume that virtual memory is always on.

Memory Management Considerations

Allocating Memory

The functions `FreeMem` (get the amount of free memory in the current heap zone), `PurgeMem` (purge blocks without compacting the heap), and `MaxMem` (compact the heap and purge all the purgeable blocks from the current heap zone) are all included in Carbon, and behave as expected under Mac OS 8/9. When your application is running on Mac OS X, however, these functions are more or less meaningless because, in Mac OS X, the system provides virtually unlimited memory.

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

Stack Size

A Carbon application may have different stack sizes under Mac OS 8/9 and Mac OS X; accordingly, the `StackSpace` function (which returns the amount of unused space on the stack at the time of the call) is no longer very useful.

Master Pointer Allocation

Master pointers do not need to be pre-allocated, or their number optimised, in the Mac OS X memory model. Accordingly, the effect of the `MoreMasterPointers` function only applies when your application is run on Mac OS 8/9.

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, dialogs 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 re-compile 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.

Note

Before the introduction of Mac OS X and Carbon, application and document resources were invariably stored in the resource fork of, respectively, application and document files. Mac OS X and Carbon introduced an alternative scheme whereby resources can be placed in the data fork of a separate resource file. The main reason for this alternative scheme is that it prevents application and document files from losing resource data when moved around different file systems or between Macintosh and non-Macintosh systems.

This alternative scheme is, however, not addressed in this book. The content of this book assumes that application and document resources are stored in the resource fork of application and document files, and the associated demonstration program files reflect that arrangement.

All Macintosh files contain both a resource fork and a data fork, even though 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 resources specific to the document, 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, typically, resources which describe the application's windows, menus, etc. Fig 4 illustrates the typical contents of the data and resource forks of an application file and a document file.

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.

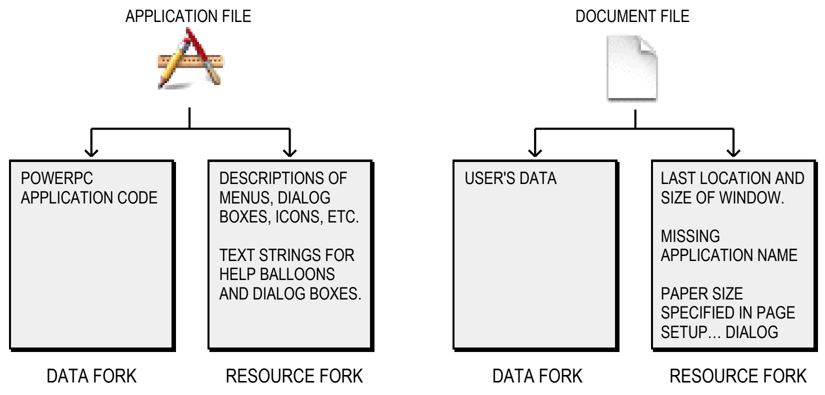
APPLICATION FILE
DOCUMENT FILE
POWERPC APPLICATION CODE
DESCRIPTIONS OF MENUS, DIALOG BOXES, ICONS, ETC.
TEXT STRINGS FOR HELP BALLOONS AND DIALOG BOXES.
USER'S DATA
LAST LOCATION AND SIZE OF WINDOW.
MISSING APPLICATION NAME
PAPER SIZE SPECIFIED IN PAGE SETUP... DIALOG
DATA FORK
RESOURCE FORK
DATA FORK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

FIG 4 - TYPICAL CONTENTS OF DATA FORKS AND RESOURCE FORKS IN APPLICATION AND DOCUMENT FILES

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 19.

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*
`'ALRT'`	Alert template.
`'DITL'`	Item list in dialog or alert.
`'FONT'`	Bitmapped font.
`'MBAR'`	Menu bar.
`'PICT'`	QuickDraw picture.

Type	*Description*
`'DLOG'`	Dialog template.
`'SIZE'`	Size of application's partition and other info.
`'WIND'`	Window template.
`'snd '`	Sound. (Space character is required.)
`'STR#'`	String list.

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.[8]

- ***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 may 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.[9] 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 5.

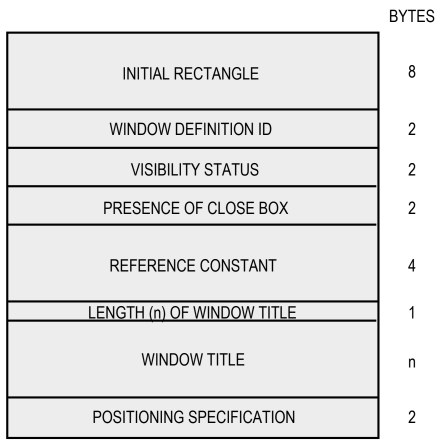

FIG 5 - STRUCTURE OF A COMPILED WINDOW ('WIND') RESOURCE

[8] 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.

[9] This book assumes the use of Resorcerer, and all demonstration program resources were created using Resorcerer.

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.

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 dialogs, 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.
- 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**.

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*	
`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 that 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 first 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 crash. (Of course, in the above code example, the possiblity of a crash is avoided because the line `if(pic1Hdl)` prevents the line from executing.)

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 `GetNewCWindow`) because functions like `GetNewCWindow` 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 block is created for an opaque data structure[10] known as a window object.
    - Data is copied from the resource template into the window object.
- The window template's block is then marked as purgeable.

Resource Manager Errors

The error code resulting from the last call to a Resource Manager function may be retrieved by calling the function `ResError`. Some of the error codes which may be returned by `ResError` are as follows:

Value	***Constant***	***Description***
`0`	`noErr`	No error occurred.
`-192`	`resNotFound`	Resource not found.
`-193`	`resFNotFound`	Resource file not found.

[10] Opaque data structures are explained at Chapter 3.

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.
```

Function

Allocating and Releasing NonRelocatable Blocks of Memory

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

Allocating and Releasing Relocatable Blocks of Memory

```
Handle  NewHandle(Size byteCount);
Handle  NewHandleClear(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    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);
```

Allocating Master Pointers

```
void    MoreMasterPointers(UInt32 inCount);
```

Accessing Memory Conditions and Freeing Memory

```
long    FreeMem(void);           // Useful under Mac OS 8/9, almost meaningless under Mac OS X.
void    PurgeMem(Size cbNeeded); // Useful under Mac OS 8/9, almost meaningless under Mac OS X.
Size    MaxMem(size *grow);      // Useful under Mac OS 8/9, almost meaningless under Mac OS X.
long    MaxBlock(void);
void    PurgeSpace(long *total,long *contig);
long    StackSpace(void);        // Not very useful in Carbon.
Size    CompactMem(Size cbNeeded);
```

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 Listing

```
// *******************************************************************************************
// SysMemRes.c
// *******************************************************************************************
//
// This program:
//
// •  Loads a window template ('WIND') resource and creates a window.
//
// •  Allocates a relocatable block in the application's heap, gets its size in bytes, draws
//    the size in the window, and then disposes of the block.
//
// •  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 Memory Manager and
//    Resource Manager functions.
//
// •  Terminates when the mouse button is clicked.
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  A 'WIND' resource (purgeable).
//
// •  A 'PICT' resource (purgeable).
//
// •  A 'STR ' resource (non-purgeable).
//
// *******************************************************************************************

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

#include <Carbon.h>

// ............................................................................... defines

#define rWindowResourceID  128
#define rStringResourceID  128
#define rPictureResourceID 128

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

void  main            (void);
void  doPreliminaries (void);
void  doNewWindow     (void);
void  doMemory        (void);
void  doResources     (void);

// ************************************************************************************ main

void  main(void)
{
  doPreliminaries();
  doNewWindow();
  doMemory();
  doResources();

  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);

  while(!Button())
   ;
}

// *************************************************************************** doPreliminaries
```

```
void  doPreliminaries(void)
{
  MoreMasterPointers(32);
  InitCursor();
}

// ******************************************************************************** doNewWindow

void  doNewWindow(void)
{
  WindowRef windowRef;

  windowRef = GetNewCWindow(rWindowResourceID,NULL,(WindowRef) -1);
  if(windowRef == NULL)
  {
    SysBeep(10);
    ExitToShell();
  }

  SetPortWindowPort(windowRef);
  UseThemeFont(kThemeSystemFont,smSystemScript);
}

// *********************************************************************************** doMemory

void  doMemory(void)
{
  Handle theHdl;
  Size   blockSize;
  OSErr  memoryError;
  Str255 theString;

  theHdl = NewHandle(1024);
  if(theHdl != NULL)
    blockSize = GetHandleSize(theHdl);

  memoryError = MemError();
  if(memoryError == noErr)
  {
    MoveTo(170,35);
    DrawString("\pBlock Size (Bytes) = ");
    NumToString(blockSize,theString);
    DrawString(theString);

    MoveTo(165,55);
    DrawString("\pNo Memory Manager errors");

    DisposeHandle(theHdl);
  }
}

// ******************************************************************************** doResources

void  doResources(void)
{
  PicHandle    pictureHdl;
  StringHandle stringHdl;
  Rect         pictureRect;
  OSErr        resourceError;

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

  SetRect(&pictureRect,148,75,348,219);
  HNoPurge((Handle) pictureHdl);
  DrawPicture(pictureHdl,&pictureRect);
  HPurge((Handle) pictureHdl);

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

  MoveTo(103,250);
  DrawString(*stringHdl);

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

  resourceError = ResError();
  if(resourceError == noErr)
  {
    MoveTo(160,270);
    DrawString("\pNo Resource Manager errors");
  }
}

// ********************************************************************************************
```

Demonstration Program SysMemRes Comments

The 'plst' (Property List) Resource

As stated in the listing, one of the resources utilised by the program is a 'plst' (property list) resource. Carbon applications must contain a 'plst' resource with ID 0 in order for the Mac OS X Launch Services Manager to recognize them as Carbon applications. If the 'plst' resource is not provided, the application will be launched as a Classic application.

The 'plst' resource used is actually an empty 'plst' resource. As will be seen at Chapter 9, however, this resource, apart from causing Mac OS X to recognise applications as Carbon applications, also allows applications to provide information about themselves to the Mac OS X Finder.

includes

Carbon greatly simplifies the matter of determining which Universal Headers files need to be included in this section in any program. All that is required is to include the header file Carbon.h, which itself causes a great many header files to be included. For this particular program, if Carbon.h is not specified, the following header files would need to be included for the reasons indicated:

Appearance.h **Constant:** kThemeSystemFont

Appearance.h itself includes MacTypes.h, which contains:

Data Types: SInt16 Str255 Size OSErr Handle Rect StringHandle
Constants: noErr NULL

Appearance.h also includes MacWindows.h, which contains:

Prototypes: GetNewCWindow GetWindowPort

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

Prototype: Button

MacWindows.h also includes Menus.h, which itself includes Processes.h, which contains:

Prototype: ExitToShell

Appearance.h also includes QuickDraw.h, which contains:

Prototypes: InitCursor SetPortWindowPort SetRect DrawPicture MoveTo GetPicture
Data Types: WindowRef PicHandle

QuickDraw.h itself includes QuickDrawText.h, which contains:

Prototypes: TextFont DrawString

MacMemory.h **Prototypes:** NewHandle DisposeHandle GetHandleSize HNoPurge HPurge MemError MoreMasterPointers

Resources.h **Prototypes:** ReleaseResource ResError

Sound.h **Prototype:** SysBeep

TextUtils.h **Prototype:** GetString

TextUtils.h itself includes NumberFormatting.h, which contains:

Prototypes: NumToString

It would be a good idea to peruse the header files MacMemory.h and Resources.h at this stage. Another header file which should be perused early in the piece is MacTypes.h.

defines

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

main

The main function calls the functions that perform certain preliminaries common to most applications, create a window, allocate and dispose of a relocatable memory block, and draw a picture and text in the window. It then waits for a button click before terminating the program

The call to the QuickDraw function QDFlushPortBuffer is ignored when the program is run on Mac OS 8/9. It is required on Mac OS X because Mac OS X windows are double-buffered. (The picture and text are drawn in a buffer and, in this program, the buffer has to be flushed to the screen by calling QDFlushPortBuffer.) This is explained in more detail at Chapter 4.

doPreliminaries

The call to MoreMasterPointers is ignored when the program is run on Mac OS X. (Master pointers do not need to be pre-allocated, or their number optimised, in the Mac OS X memory model.)

On Mac OS 8/9, the call to MoreMasterPointers to allocate additional master pointers is really not required in this simple program because the 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 to MoreMasterPointers 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. Note that specifying a number less than 32 in the inCount parameter will result in 32 master pointers in the allocated block.

The call to InitCursor sets the cursor shape to the standard arrow cursor and sets the cursor level to 0, making it visible.

doNewWindow

doNewWindow creates a window.

The call to GetNewCWindow creates a window using the 'WIND' template resource specified in the first parameter. The type, size, location, title, and visibility of the window are all established by the 'WIND' resource. The third parameter tells the Window Manager to open the window in front of all other windows.

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

The call to SetPortWindowPort makes the new window's graphics port the current port for drawing operations. The call to the Appearance Manager function UseThemeFont sets the font for that port to the system font.

doMemory

doMemory allocates a relocatable block of memory, gets the size of that block and draws it in the window (or, more accurately, in the window's graphics port), and checks for memory errors.

The call to NewHandle allocates a relocatable block of 1024 bytes. The call to GetHandleSize returns the size of the allocated area available for data storage (1024 bytes). (The actual memory used by a relocatable block includes a block header and up to 12 bytes of filler.)

The call to MemError returns the error code of the most recent call to the Memory Manager. If no error occurred, the size returned by GetHandleSize is drawn in the window, together with a message to the effect that MemError returned no error. DisposeHandle is then called to free up the memory occupied by the relocatable block.

MoveTo moves a "graphics pen" to the specified horizontal and vertical coordinates. DrawString draws the specified string at that location. NumToString creates a string of decimal digits representing the value of a 32-bit signed integer. These calls are incidental to the demonstration.

doResources

doResources 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 window.

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 AND C 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):

```
+---+---+---+---+---+---+---+---+---+---+
| M | Y |   | S | T | R | I | N | G | 0 |
+---+---+---+---+---+---+---+---+---+---+
```

A C string is thus said to be "null-terminated".

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

```
+---+---+---+---+---+---+---+---+---+---+
| 9 | M | Y |   | S | T | R | I | N | G |
+---+---+---+---+---+---+---+---+---+---+
```

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

In the demonstration program, you encoutered 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 MacTypes.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 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 (CopyPascalStringToC, CopyCStringToPascal) 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 doResources with the following, compile-link-run, and note the way the second and third strings appear in the window.

```
    void  doResources(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 reference
    }
```

MORE ON STRINGS — UNICODE AND CFString OBJECTS

In the Carbon era, no discussion of strings would be complete without reference to Unicode and CFString objects.

Preamble — Writing Systems, Character Encoding and Character Sets,

A writing system (eg., Roman, Hebrew, Arabic) comprises a set of characters and the basic rules for using those characters in creating a visual depiction of a language. An individual character is a symbolic representation of an element of a writing system.

In memory, an individual character is stored as a character code, a numeric value that defines that particular character. One byte (8 bits) is commonly used to store a single character code, which allows for a character set of 256 characters maximum.

A total of 256 numeric values is, of course, quite inadequate to provide for all the characters in all the world's writing systems, meaning that different character sets must be used for different writing systems. In one-byte encoding systems, therefore, these different character sets are said to "overlap".

As an aside, the Apple Standard Roman character set (an extended version of the 128-character ASCII character set) is the fundamental character set for the Macintosh computer. To view the printable characters in this set, replace the main function in the SysMemRes demonstration program with the following:

```
void  main(void)
{
  WindowRef windowRef;
  SInt16    fontNum, a,b;
  UInt8     characterCode = 0;

  doPreliminaries();
  windowRef = GetNewCWindow(rWindowResourceID,NULL,(WindowPtr) -1);
  SetPortWindowPort(windowRef);

  GetFNum("\pGeneva",&fontNum);
  TextFont(fontNum);

  for(a = 0;a < 256;a += 16)
  {
    for(b=0;b<16;b++)
    {
      MoveTo((a * 1.5) + 60,(b * 15) + 40);
      DrawText(&characterCode,0,1);
      characterCode++;
    }
  }

  while(!Button())
   ;
}
```

In addition to the overlapping of character sets between writing systems, a further problem is conflicting character encodings within a single writing system. For an example of this in the Roman writing system, change "\pGeneva" to "\pSymbol" in the above substitute main function.

Unicode

Unicode is an international standard that combines all the characters for all commonly used writing systems into a single character set. It uses 16 bits per character, allowing for a character set of up to 65,535 characters. With Unicode, therefore, the character sets of separate writing systems do not overlap. In addition, Unicode eliminates the problem of conflicting character encodings within a single writing system, such as that between the Roman character codes and the codes of the symbols in the Symbol font.

Unicode makes it possible to develop and localize a single version of an application for users who speak most of the world's written languages, including Russian (Cyrillic), Arabic, Chinese, and

Core Foundation's String Services

Core Foundation is a component of the system software. Through its String Services, Core Foundation facilitates easy and consistent internationalization of applications. The essential part of this support is an opaque data type (CFString). A CFString "object" represents a string as an array of 16-bit Unicode characters. Several Appearance Manager, Control Manager, Dialog Manager, Window Manager, Menu Manager, and Carbon Printing Manager functions utilize CFStrings.

CFString objects come in immutable and mutable variants. Mutable strings may be manipulated by appending string data to them, inserting and deleting characters, and padding and trimming characters.

CFStrings have many associated functions that do expected things with strings such as comparing, inserting, and appending. Functions are also available to convert Unicode strings (that is, CFStrings) to and from other encodings, particularly 8-bit encodings (such as the Apple Standard Roman character encoding) stored as Pascal and C strings.

Example

For a basic example of the use of CFStrings, remove the calls to doMemory and doResources, and the functions themselves, from the main function in the original version of the demonstration program SysMemRes and replace the function doNewWindow with the following:

```
void  doNewWindow(void)
{
  WindowRef   windowRef;
  Str255      pascalString = "\pThis window title was set using a CFString object";
  CFStringRef titleStringRef;
  CFStringRef textStringRef = CFSTR("A CFString object converted to a Str255");
  Boolean     result;
  Str255      textString;

  windowRef = GetNewCWindow(rWindowResourceID,NULL,(WindowPtr) -1);
  SetPortWindowPort(windowRef);
  UseThemeFont(kThemeSystemFont,smSystemScript);

  titleStringRef = CFStringCreateWithPascalString(NULL,pascalString,
                                                  CFStringGetSystemEncoding());
  SetWindowTitleWithCFString(windowRef,titleStringRef);

  result = CFStringGetPascalString(textStringRef,textString,256,
                                   CFStringGetSystemEncoding());
  if(result)
  {
    MoveTo(135,130);
    DrawString(textString);
  }

  if(titleStringRef != NULL)
    CFRelease(titleStringRef);
}
```

At the sixth line, an immutable CFString object is created. The easiest way to do this is to use the CFSTR macro. The argument of the macro must be a constant compile-time string (that is, text enclosed in quotation marks) that contains only ASCII characters. CFSTR returns a reference (textStringRef) to a CFString object. This will be used later.

The call to the String Services function CFStringCreateWithPascalString converts a Pascal string (pascalString, of type Str255) to a CFString object. The reference to the CFString object is then passed in a call to the Window Manager function SetWindowTitleWithCFString to set the window's title.

The call to the String Services function CFStringGetPascalString converts the CFString object (textStringRef) previously created by the CFSTR macro to a Pascal string (textString, of type Str255). 256, rather than 255, is passed in the bufferSize parameter to accommodate the length byte. textString is then passed in a call to the QuickDraw function DrawString, which draws the string in the window.

The golden rules for releasing CFString objects are: if a "Create" or "Copy" function is used, CFString should be called to release the string when no longer required; if a "Get" function is used, CFString should not be called. Accordingly, CFRelease is called only in the case of titleStringRef. Note that CFRelease is not NULL-safe, so you must check for a non-NULL value before passing something to CFRelease.