To:	Cortland Software Group Software ERS binder
From:	Rich Williams
Subject:	**Indiana Jones and the Memory Manager of Doom**

Revision history

Rev 1 Nov. 5, 1985 First pass

Rev 2 Nov. 27, 1985

PurgeAll, Lockall, etc. added. Properties of blocks added.

Rev 3 Feb. 10, 1986

Copy commands added. Call numbers added. The order of parameters changed. Error codes added. TotalMem call added.

Rev 4 Mar. 10,1986

Standard calls added. Parameters added to AppInit and AppQuit. SetPurgeAll parameters switched.

Rev 5 July 14, 1986

CheckHandle, CompactMem, PtrToHand, HandToPtr and HandToHand and RestoreHandle calls added. Section 2, Design Philosophy, removed . New section 2 added. Section 7 added.

Rev 6 July 29,1986

Locked handles can be disposed. PurgeAll only purges unlocked handles. These were typos in earlier revisions. ReAllocHandle parameters were in wrong order. Error codes for calls added. Section 2.5.1 added. Warning added in section 5.1. NilErr changed to EmptyErr, NotNilErr changed to NotEmptyErr to avoid confusion between handles that are Nil and handles containing Nil.

Table of Contents

1.0 Introduction

'Memory [management] is the treasury and guardian of all things'
Cicero, De Oratore. Bk. i, Sec. 5

'The memory [manager] strengthens as you lay burdens upon it, and becomes trustworthy as you trust it.'
Thomas De Quincy, Confessions on an English Opium-Eater

The memory manager is the bookkeeper for the memory in the Apple //. By using the memory manager, programs can dynamically allocate, deallocate and resize memory blocks in any order according to their needs. The memory manager keeps track of the owner of each memory block so that more than one program can share the available ram in the Apple. A desk accessory, for example, can ask for memory even though it is being called while running an application.

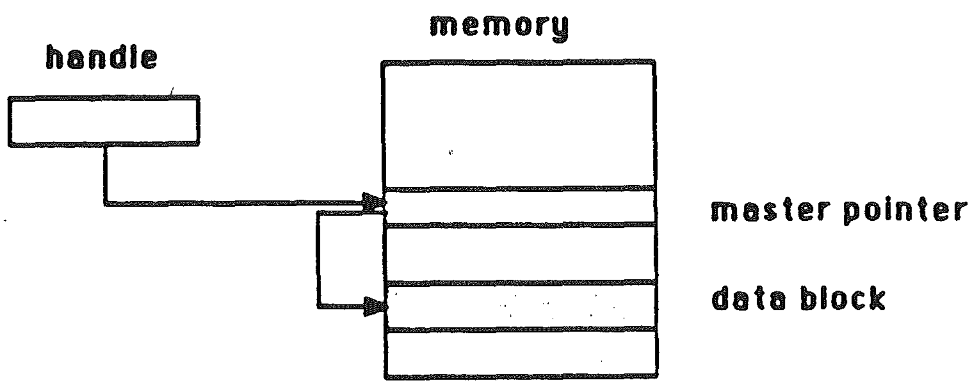
Conceptually, the memory manager is very similar to the Macintosh. The user, however, should not be lulled into thinking that it is compatible with the Macintosh's. **Because of the architecture of the Apple // and the 65816, the calls are very different and the internal data structures are totally different from the Macintosh.**

2.0 Fundamental concepts

In order to understand how the memory manager works, there are some basic concepts that need to be understood. These are handles, master pointers, fragmentation, compaction and purging. The use of these terms are almost identical to the Macintosh.

2.1 Handles and master pointers

When a new block is created for a program, the program is not given the address of the actual block. Instead, the program is given the address of a pointer to the block. This address is called the **handle** and the pointer to the block is called the **master pointer**. This way, if the block is moved, the program can still find the block by looking at the master pointer, which never moves. The following picture illustrates the indirection through the master pointer.



Handle points to master pointer which points to actual block.

Figure xx. Handle to a Relocatable Block

In order to access the information in the block, the program must copy the starting address from the master pointer into zero page to use indirect addressing. This is called **dereferencing** the handle. Once this is done, the program must make sure that the block doesn't move. The block might be moved if the memory manager is called or a routine that calls the memory manager is called. If the block is moved, it must be dereferenced again.

2.2 Fixed and relocatable blocks

Some blocks must never be moved. 6502 code, for example, is seldom position independent and if moved will no longer work. Blocks of data can usually be moved without harm. The memory manager allows **fixed** blocks to be created for position dependent data (i.e. programs) and **relocatable** blocks to be created for position independent data that can be moved when necessary. The memory manager tries, but does not guarantee, to allocate fixed blocks in low memory and relocatable blocks in high memory.

2.3 Memory fragmentation

Since memory blocks can be allocated and deallocated in any order, memory tends to become fragmented after a while into a jumble of free and allocated memory blocks. When this happens, the memory manager may not be able to allocate a requested block even though there is enough free memory available because the space is broken up into smaller, isolated blocks. The following picture illustrates fragmented memory.

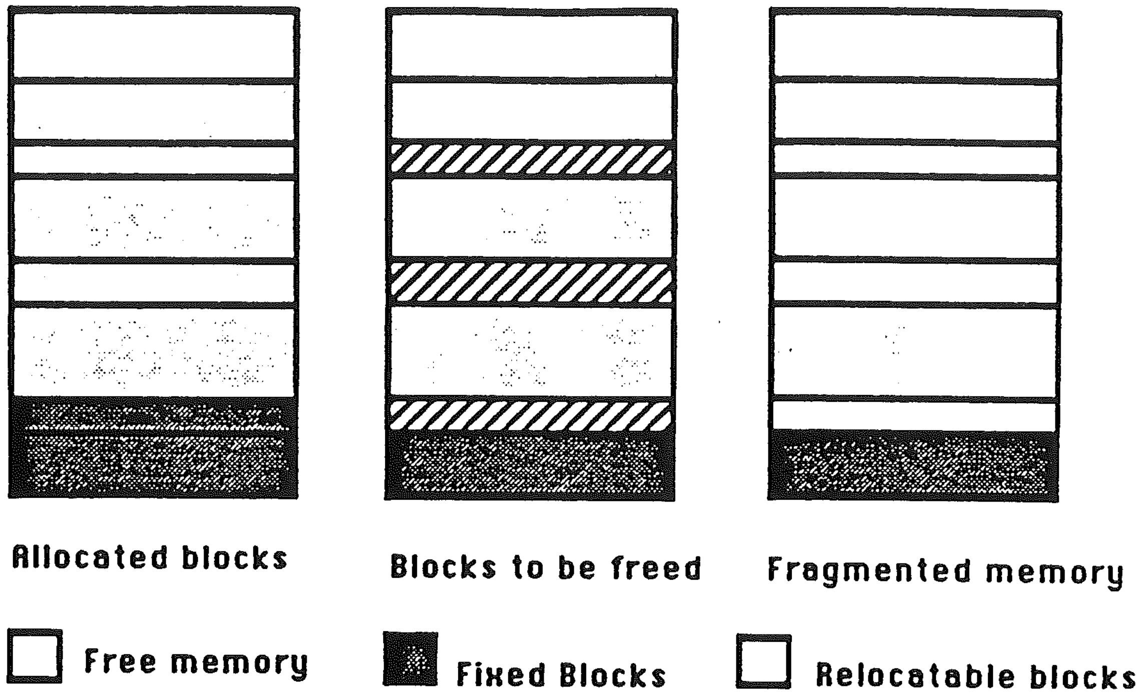


Figure xx. Memory Fragmentation

2.4 Compaction

When the memory manager is unable to allocate a block it will try to **compact** memory. Compaction is moving all of the relocatable blocks to consolidate the free space into a single block. Compaction of the above example is shown in figure XX.

Fixed blocks and locked relocatable blocks interfere with compaction by forming immovable islands in memory. This can prevent the free blocks from being collected together and can leave memory fragmented after compaction, as shown in figure XX. The memory manager never moves a relocatable block around a non-relocatable one. To minimize this problem, the memory manager tries to allocate fixed blocks towards the bottom of memory and relocatable blocks towards the top of memory. Also, to help prevent fragmentation, programs should use relocatable blocks whenever possible and leave blocks unlocked as much as possible.

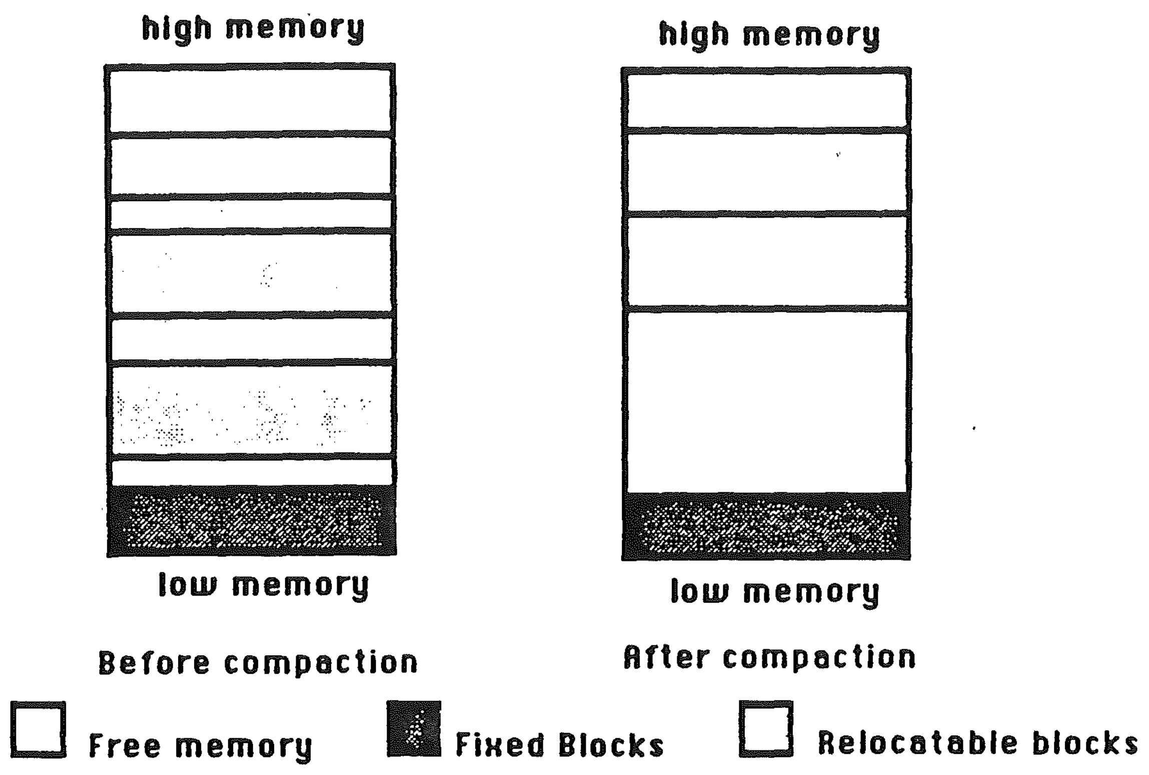


Figure xx. Memory Compaction

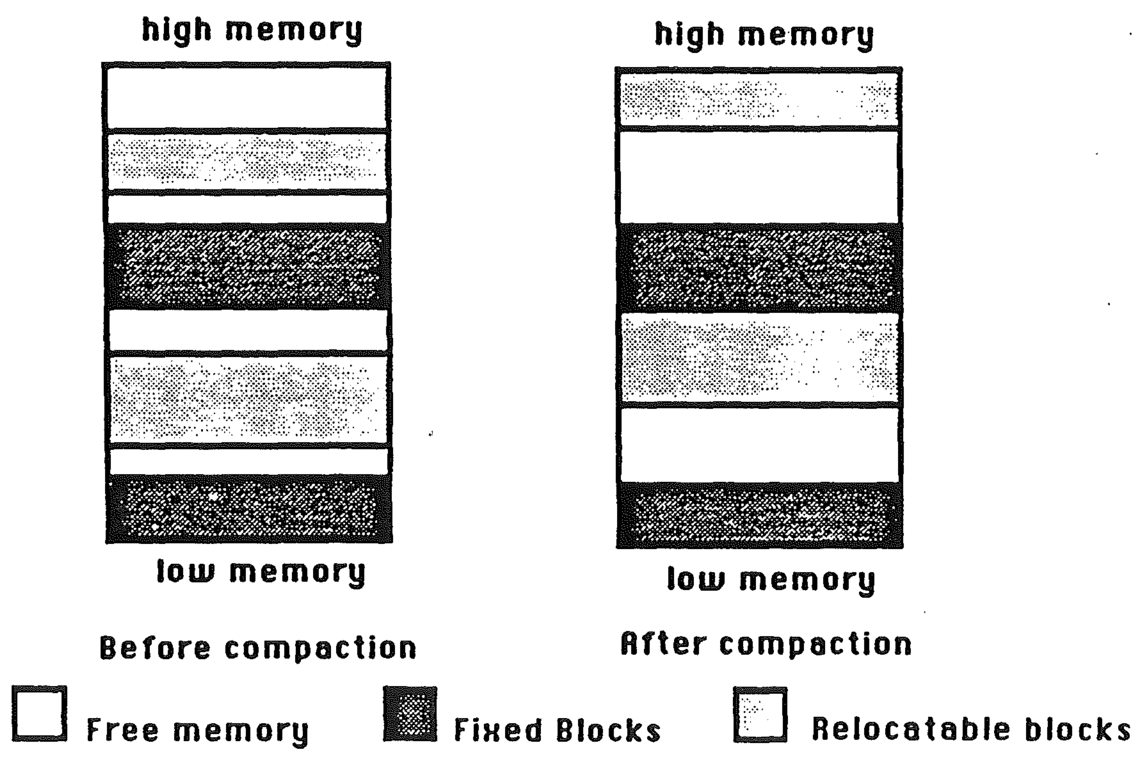


Figure xx. Fragmentation after Compaction

2.5 Purging

If the memory manager is still unable to allocate a block after compacting memory, it will try **purging** blocks. Only blocks that are marked purgable and unlocked can be purged. Purging throws out the contents of the block and frees it. THe block's master pointer remains allocated and its value is set to nil. A handle pointing to nil master pointer is called an **empty handle**. If your programs wants to refer to the purged block, it must detest that the handle is empty and ask the memory manager to reallocate the block. The data in the block has been lost and must be recreated by the program. Figure XX shows a block being purged and reallocated.

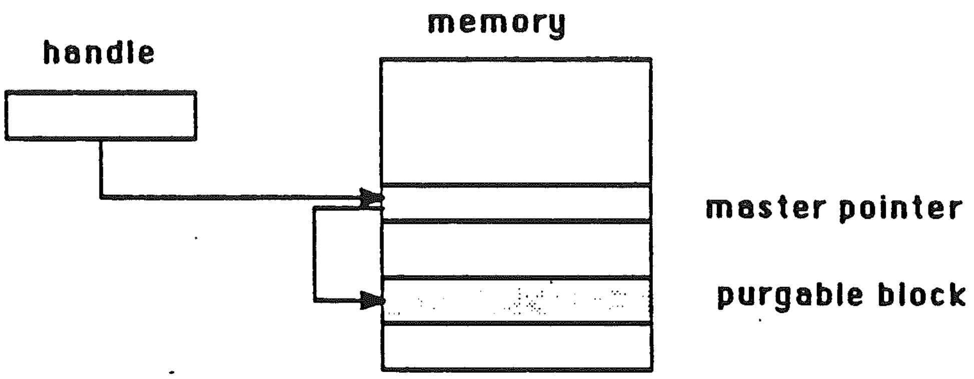


Figure XX a. Before purging.

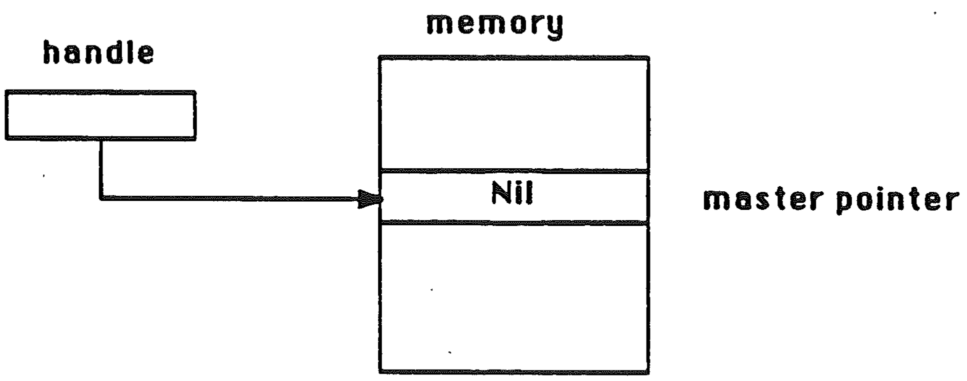


Figure XX b. After purging. Handle is empty and data is lost.

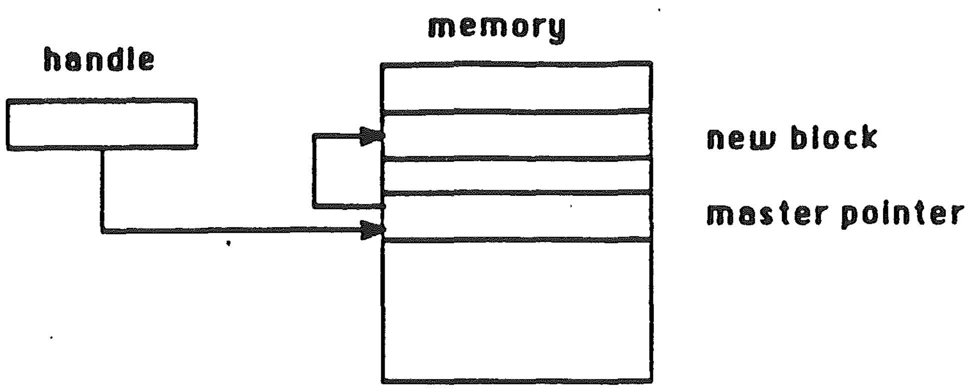


Figure XX c. Pointer points to new block.

2.5.1 Automatic Purging

When the memory manager runs out of memory, it will start purging purgable blocks to attempt to make more room. The order of the purging is based on the **Purge Level** of the block. The purge level is a 2 bit number specifying the purging priority of the block. The values are:

3:	Most purgable. Used for putting programs in zombie state (see below).
2:	Next most purgable.
1:	Least purgable.
0:	Not purgable.

Level 3 is used by the system loader. When some applications are exited, the memory is not freed but its blocks are set to level 3. This way the old application can be restarted later without going to the disk if the new application did not need the space. The old application is in what is called the zombie state. If the memory manager purges any blocks of an application in the zombie state, it will purge all of the blocks. A application should only use levels 0 to 2.

2.6 Special Memory

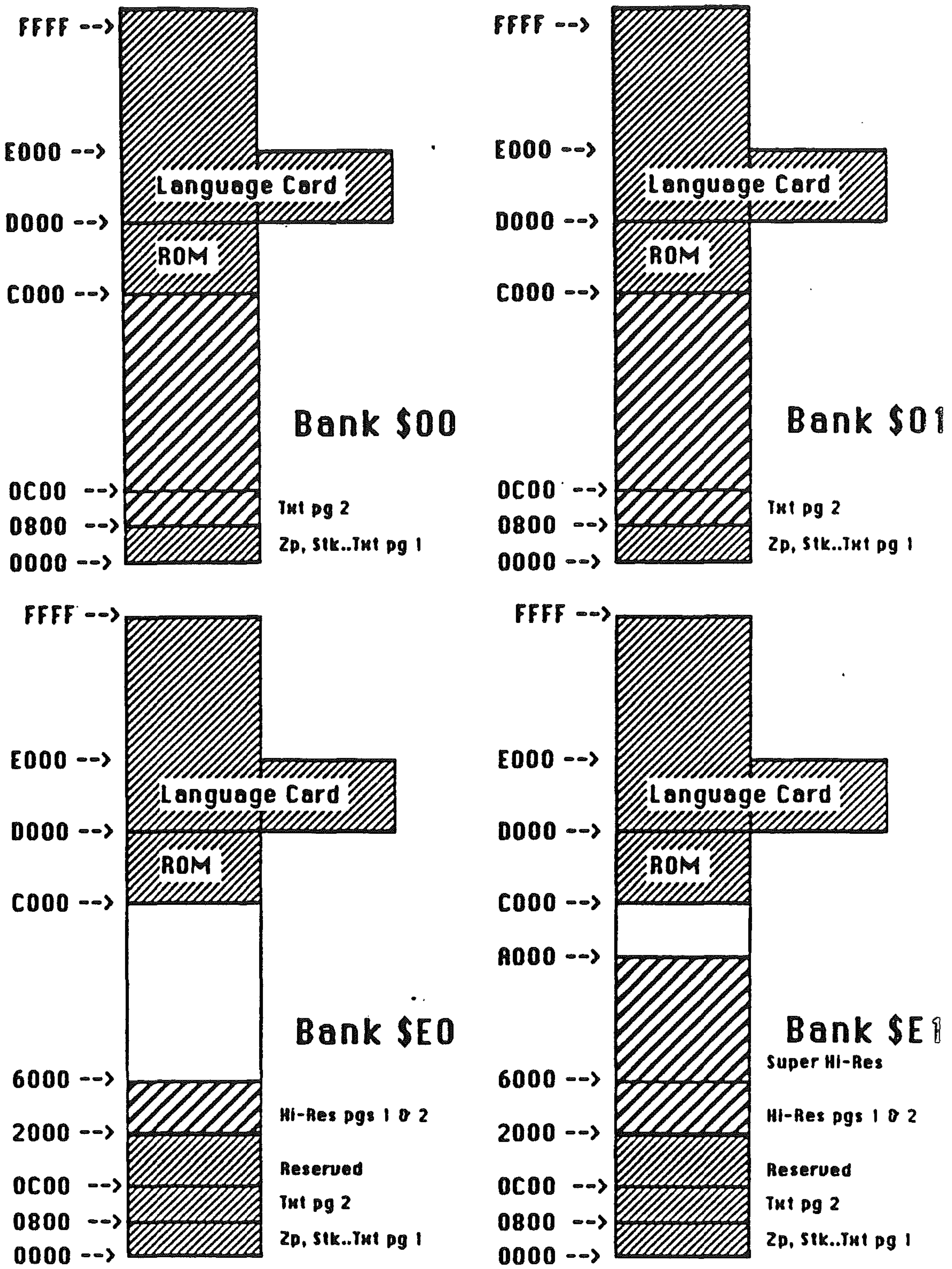
The memory in the Apple // is divided into three catagories. They are:

1) Nonspecial or normal memory - This is memory that has no special restrictions on it. Banks 2 - $DF and parts of banks $E0 and $E1.

2) Special memory - This is memory that has restrictions on its use because it is memory that appears in the Apple //e. Special memory may not be used by desk accessories, tools and other routines that might be called while running old applications. Banks 0 - 1 and parts of banks $E0 and $E1 are special memory.

3) Reserved memory - This is memory that is not managed by the memory manager. This includes the language cards, addresses $0000 - $0800 in banks 0 and 1, and addresses $0000 - $2000 in banks $E0 and $E1. This memory is marked busy in the memory manager at startup time.

How the Memory Manager Uses Banks $00, $01, $E0, $E1



Unmanaged Memory ▨	Allocatable, but Special ▨ (//e+video memory)
Allocatable Memory ☐	

3.0 Differences from the Mac memory manager

There are many differences between this memory manager and the Macintosh's. Some of the most notable are:

Fewer calls. If you want others, ask for them
Blocks now have an owner ID
There is now a purging priority level for the block
There are only handles and no pointers

4.0 Properties of memory blocks

Memory blocks have attributes that determine how they are allocated and maintained. Some attributes are defined at allocation time and can't be changed. Others can be modified after allocation.

4.1 Allocation attributes

The memory in the Apple // and the architecture of the 65816 force many restrictions on how blocks can be allocated. Blocks, for example, may have to be page aligned or they may have to be in a certain bank. When allocating a block, an attributes word is specified that determines how the block is allocated. These attributes can only be set when the block is allocated. The attributes are:

D14	Fixed
D4	May not cross bank boundary
D3	May not use special memory
D2	Page Aligned
D1	Fixed Address
D0	Fixed Bank

Dn = bit in attributes word. D15 = msb D0 = lsb

Fixed:

If a block is fixed, it cannot be moved when compacting memory. Code blocks will usually be fixed, but data blocks should usually not be.

May not cross a bank boundary:
This specifies that a block must not cross banks. Code blocks, for example, may never cross banks.

May not use special memory:
This specifies that the block may not be allocated in special memory. This is memory that is used in the Apple //e and includes banks 0, 1 and the video screens.

Page aligned:
For timing reasons, code or data may need to be page aligned

Fixed Address:
This is used to specify that the block must be at a specified address when allocated. An example is allocating the graphics screen.

Fixed Bank:
This specifies that the block must start in a specified bank. An example is allocating a block to be used as a zero page or stack.

4.2 Modifiable attributes

The memory manager can move or purge a block while making room for a new block. There are attributes that determine whether a block can be moved or purged. These attributes can be changed by the user after a block is created. The attributes are:

D15	Locked
D9.8	PurgeLevel

Locked:
When a block is locked, it is unmovable and unpurgeable irregardless of what Movable or PurgeLevel is set to. This feature is to allow a block to be temporily locked down while it is being executed or referenced.

PurgeLevel:
This is a two bit number defining the purging priority of a block. 0 is unpurgable and level 3 is the first purged. Applications should normally use levels 0 to 2.

5.0 Memory Manager Calls

Calls to the memory manager fall into the following catagories:

Standard calls		
$01	MMBootInit	Boot time initialization
$02	MMAppInit	Application initialization
$03	MMAppQuit	Application quit call
$04	MMGetVersion	Gets version number
$05	MMReset	Called by system reset
$06	MMStatus	Active status
Allocating memory		
$09	NewHandle	Creates a new block and handle
$0A	ReallocHandle	Uses an existing handle
$0B	RestoreHandle	Restores a purged handle
Freeing memory		
$10	DisposHandle	Deallocates a handle
$11	DisposAll	Deallocates all of an owner's memory
$12	PurgeHandle	Purges the contents of a block
$13	PurgeAll	Purges all of an owner's purgable blocks
Information on blocks and memory		
$18	GetHandleSize	Gets the size of a block
$19	SetHandleSize	Grows or shrinks a block
$1A	FindHandle	Finds the handle of a block containing an address
$1B	FreeMem	Gets total amount of free space
$1C	MaxBlock	Gets size of largest free block
$1D	TotalMem	Gets size of all memory
$1E	CheckHandle	Checks if a handle is valid
$1F	CompactMem	Forces memory compaction
Other properties of blocks		
$20	HLock	Locks a block
$21	HLockAll	Locks all of an owners blocks
$22	HUnlock	Unlocks a block
$23	HunlockAll	Unlocks all of an owner's blocks
$24	SetPurge	Sets the purge level of a block

$25	SetPurgeAll	Sets the purge level of all of an owner's blocks
Copying Data		
$28	PtrtoHand	Copies from an address to a handle
$29	HandtoPtr	Copies from a handle to an address
$2A	HandtoHand	Copies from one handle to another
$2B	BlockMove	Copies from one address to another

5.1 Data types

These are the data types used in the calls:

Pointer	= ^Byte	
Handle	= ^Pointer	
UserID	= Word	{Identifies. the owner of a segemnt}
Address	= Long int	{4 byte address}
Size	= Long int	{4 byte size of a block}
PurgeLevel	= 0..3	{Priority to purge a block}

While the 65816 has only a 24 bit address space, addresses are always given as 32 bit (4 byte) values with the high byte 0. Programs should never attempt to store other information in the high byte of the address. If you do, the memory manager and other tools may not work properly.

5.2 Standard calls

These are standard calls defined for every tool. Note that the AppInit call is different from other tool sets.

5.2.1 MMBootInit

This call initializes the memory manager at boot time. An application must never make this call since it will destroy all currently allocated blocks including the caller. Never, ever, ever make this call. Don't even try to use it as part of a protection scheme.

Possible errors:

Code	Type of error

$0000	No error

5.2.2 MMAppInit

inputs:	none	
output:	Owner:	UserID

This call is made by an application when it starts up. If the call is not made from a valid segment, a IDErr is returned. If this happens, the program should call the ID Manager for a ID number and then call the memmory manager to allocate its program segments. This should only happen when running under the current operating systems.

Possible errors:

<u>Code</u>	<u>Type of error</u>	
$0000	No error	
$0207	IDErr	An invalid owner ID was given

5.2.3 MMAppQuit

inputs:	Owner:	UserID
output:	none	

This call is given to the memory manager by the application when it is finished and is about to exit.

Possible errors:

<u>Code</u>	<u>Type of error</u>
$0000	No error

5.2.4 MMGetVersion

inputs:	none	
output:	Version:	word

This returns the version number of the memory manager.

Possible errors:

Code	Type of error
$0000	No error

5.2.5 MMReset

The memory manager will check the internal lists and return a MemErr if they are inconsistent. This is an internal call used by the system at reset time. An application should never make this call since future memory managers may attempt to clean up a damaged system. This could destroy the application.
Possible errors:

Code	Type of error	
$0000	No error	
$0201	MemErr	Memory lists damaged.

5.2.6 MMStatus

inputs:	none		
outputs:	Status:	Boolean	(always true)

Status is used to test if the tool is active. The memory manager is always active.
Possible errors:

Code	Type of error
$0000	No error

5.3 Allocating memory

These commands are used to create memory blocks.

5.3.1 NewHandle

inputs:	BlockSize:	Size
	Owner:	UserID
	Attributes:	Word
	Location:	Address
outputs:	Handle	

NewHandle is used to create a new block. BlockSize is the size of the block to create. The attributes are described in section 4. If a block of size 0 is created, the handle will be set to NIL. If a block of size 0 is created, it must be unlocked and movable.

Possible Errors:

Code	Type of error	
$0000	No error	
$0201	MemErr	Unable to allocate block.
$0204	LockErr	Illegal operation on a locked or immovable block
$0207	IDErr	An invalid owner ID was given

5.3.2 ReallocHandle

inputs:	BlockSize:	Size
	Owner:	UserID
	Attributes:	Word
	Location:	Address
	TheHandle:	Handle
output:	none	

ReallocHandle is used to reallocate a block that has been purged. BlockSize is the size of the block to create. The attributes are described in section 4 Any information that was in the purged block has been lost.

Possible Errors:

Code	Type of error	
$0000	No error	
$0201	MemErr	Unable to allocate block.
$0203	NotEmptyErr	An empty handle was expected for this operation
$0204	LockErr	Illegal operation on a locked or immovable block
$0206	HandleErr	An invalid handle was given
$0207	IDErr	An invalid owner ID was given

5.3.2 RestoreHandle

inputs: TheHandle: Handle

output: none

RestoreHandle is also used to reallocate a block that has been purged. RestoreHandle will use the same attributes, owner and size that were in the purged handle. The block may not be fixed address or fixed bank. If it is, an AttrErr will be returned. Any information that was in the purged block has been lost.

Possible errors:

Code	Type of error	
$0000	No error	
$0201	MemErr	Unable to allocate block.
$0203	NotEmptyErr	An empty handle was expected for this operation
$0206	HandleErr	An invalid handle was given
$0208	AttrErr	Operation illegal on block with given attributes.

5.4 Freeing memory

These commands are used to free blocks and pointers. Once a block or handle is freed, its contents cannot be recovered.

5.4.1 DisposeHandle

inputs:	theHandle:	Handle
output:	none	

DisposHandle purges the block specified by theHandle and deallocates the handle. The block's purge level and locked status are ignored.

Possible Errors:

Code	Type of error	
$0000	No error	
$0206	HandleErr	An invalid handle was given

5.4.2 DisposeAll

inputs:	Owner:	UserID
output:	none	

DisposAll disposes all of the handles owned by Owner.

Possible Errors:

Code	Type of error	
$0000	No error	
$0207	IDErr	An invalid owner ID was given

5.4.3 PurgeHandle

inputs:	theHandle:	Handle
output:	none	

PurgeHandle purges the block specified by theHandle. The block must be purgable and unlocked. The handle itself remains allocated but is empty (pointed to NIL).

Possible Errors:

Code	Type of error	
$0000	No error	
$0204	LockErr	Illegal operation on a locked or immovable block
$0205	PurgeErr	Attempt to purge an unpurgable block
$0206	HandleErr	An invalid handle was given

5.4.4 PurgeAll

inputs:	Owner:	UserID
output:	none	

PurgeAll purges all of the purgable blocks owned by Owner. Only purgable, unlocked blocks are purged. If any blocks were not purgable, LockErr or PurgeErr will be returned and the purgable blocks will be purged.

Possible Errors:

Code	Type of error	
$0000	No error	
$0204	LockErr	Illegal operation on a locked or immovable block
$0205	PurgeErr	Attempt to purge an unpurgable block
$0207	IDErr	An invalid owner ID was given

5.5 Information on Blocks

These commands are used to grow or shrink memory blocks.

5.5.1 GetHandleSize

inputs:	theHandle:	Handle
output:	Size	

GetHandleSize returns the size of a block specified by theHandle.

Possible Errors:

Code	Type of error	
$0000	No error	
$0206	HandleErr	An invalid handle was given

5.5.2 SetHandleSize

inputs:	newSize:	Size
	theHandle:	Handle
output:	none	

SetHandleSize changes the size of the block specified by theHandle. The block can be made larger or smaller. If necessary to lengthen a block, memory may be compacted or blocks may be purged. The handle should be unlocked since it may have to move to change size. If the size is set to 0, the handle will be set to NIL. Attempting to resize a purged handle will return a EmptyErr.

Possible Errors:

Code	Type of error	
$0000	No error	
$0201	MemErr	Unable to allocate block.
$0202	EmptyErr	Illegal operation on An empty handle
$0204	LockErr	Illegal operation on a locked or immovable block
$0206	HandleErr	An invalid handle was given

5.5.3 FindHandle

inputs:	Location:	Address
output:	theHandle:	Handle

FindHandle returns the handle to the block containg the address specified by location. Note that if the block is not locked, it may move. If the address is not in any handle, then NIL (0) is returned.

Possible Errors:

Code	Type of error
$0000	No error

5.5.4 FreeMem

inputs: none

output: Size

FreeMem returns the total number of free bytes in memory. It does not count memory that could be freed by purging. Because of memory fragmentation, it may not be possible to allocate a block this large. FreeMem does a compaction of the memory space.

Possible Errors:

Code	Type of error
$0000	No error

5.5.5 MaxBlock

inputs: none

output: size

MaxBlock returns the size of the largest free block in memory. It does not count memory that could be freed by purging or compacting.

Possible Errors:

Code	Type of error
$0000	No error

5.5.6 TotalMem

inputs: none

output: size

TotalMem returns the size off all of the memory in the machine. This includes the main 256K.

Possible Errors:

Code	Type of error
$0000	No error

5.5.7 CheckHandle

inputs: theHandle: Handle

outputs: none

CheckHandle checks to see if a handle is a valid handle.. This call is intended primarily as a debugging aid. If the memory manager does not recognize the handle as one it created, HandleErr is returned.

Possible errors:

Code	Type of error	
$0000	No error	
$0206	HandleErr	An invalid handle was given

5.5.7 CompactMem

inputs: none

outputs: none

CompactMem can be used to force memory compaction. Memory compaction is never done during interrupts so if CompactMem is called from an interrupt, no compaction is done.

Possible errors:

Code	Type of error
$0000	No error

5.6 Other properties of blocks

These commands change the other properties of memory blocks.

5.6.1 HLock

inputs:	theHandle:	Handle
output:	none	

HLock locks a block specified by theHandle. A locked block cannot be relocated or purged during memory compaction.

Possible Errors:

Code	Type of error	
$0000	No error	
0206	HandleErr	An invalid handle was given

5.6.2 HLockAll

inputs:	Owner:	UserID
output:	none	

HLockAll locks all of the blocks owned by Owner.

Possible Errors:

Code	Type of error	
$0000	No error	
$0207	IDErr	An invalid owner ID was given

5.6.3 HUnlock

inputs:	theHandle:	Handle
output:	none	

HUnlock unlocks a block specified by theHandle. A unlocked block can be relocated during memory compaction.

Possible Errors:

Code	Type of error	
$0000	No error	
$0206	HandleErr	An invalid handle was given

5.6.4 HUnlockAll

inputs:	Owner:	UserID
output:	none	

HUnlockAll unlocks all of the blocks owned by Owner.

Possible Errors:

Code	Type of error	
$0000	No error	
$0207	IDErr	An invalid owner ID was given

5.6.5 SetPurge

inputs:	newPlevel:	PurgeLevel (word)
	theHandle:	Handle
output:	none	

SetPurge sets the PurgeLevel of the block specified by theHandle to newPlevel.

Possible Errors:

Code	Type of error	
$0000	No error	
$0206	HandleErr	An invalid handle was given

5.6.6 SetPurgeAll

inputs:	newPlevel:	PurgeLevel (word)
	Owner:	UserID
output:	none	

SetPurgeAll sets the purge level of all of the blocks owned by Owner.

Possible Errors:

Code	Type of error	
$0000	No error	
$0207	IDErr	An invalid owner ID was given

5.7 Copying Data

These commands are used to copy data from one place to another in the machine. The moves will work properly even if the source and destination blocks overlap or cross bank boundaries. All of these functions are essentially the same as block move. PtrtoHand, HandtoPtr and HandtoHand will dereference the handles for the caller's convience. The calls do not verify that a destination block is large enough to hold the data. There is no address validation on pointers and the functions will cheerfully write over anything even if it crashes the machine. The high byte of the four byte address must be 0.

5.7.1 PtrtoHand

inputs:	Source:	Pointer
	Dest:	Handle
	Count:	Size
output:	none	

Possible Errors:

Code	Type of error	
$0000	No error	
$0202	EmptyErr	Illegal operation on An empty handle
$0206	HandleErr	An invalid handle was given

5.7.2 HandtoPtr

inputs:	Source:	Handle
	Dest:	Pointer
	Count:	Size
output:	none	

Possible Errors:

Code	Type of error	
$0000	No error	
$0202	EmptyErr	Illegal operation on an empty handle
$0206	HandleErr	An invalid handle was given

5.7.3 HandtoHand

inputs:	Source:	Handle
	Dest:	Handle
	Count:	Size
output:	none	

Possible Errors:

<u>Code</u> <u>Type of error</u>

$0000	No error	
$0202	EmptyErr	Illegal operation on an empty handle
$0206	HandleErr	An invalid handle was given

5.7.4 BlockMove

inputs:	Source:	Pointer
	Dest:	Pointer
	Count:	Size
output:	none	

Possible Errors:

Code	Type of error
$0000	No error

6.0 Error Codes

These are the error codes returned by the memory manager

Code	Type of error	
$0000	No error	
$0201	MemErr	Unable to allocate block.
$0202	EmptyErr	Illegal operation on an empty handle
$0203	NotEmptyErr	An empty handle was expected for this operation
$0204	LockErr	Illegal operation on a locked or immovable block
$0205	PurgeErr	Attempt to purge an unpurgable block
$0206	HandleErr	An invalid handle was given
$0207	IDErr	An invalid owner ID was given (usually zero)
$0208	AttrErr	Operation illegal on block with given attributes.

7.0 ROM and Ram portions of the memory manager

All of the memory manager is in rom except for the automatic purging routines. The first release of the memory manager has these in ram to allow the operating syatem and loader people to determine the proper purging priorites. Future releases of the rom will have these routines in rom.