New Technical Notes

Macintosh

Developer Support

ME 13 - Memory Manager Compatibility

Memory

Written by: Jeff Crawford June 1993

The Memory Manager has been rewritten for the PowerPC Macintosh computers. This new Memory Manager runs native PowerPC code and uses better algorithms. With this new Memory Manager, there are both old and new restrictions on it use.

Topics

- A list of things not to do
- A new list of restrictions for being compatible in the future
- A list of recommendations

Introduction

The Memory Manager has been rewritten for Apple's PowerPC Macintosh computers. The new Memory Manager uses native PowerPC code and better algorithms for faster performance. It is also not as forgiving to ruthless programmers who break the rules defined in *Inside Macintosh* and Macintosh Technical Notes.

First a conceptual clarification. The Memory Manager as documented in Inside Macintosh is a storage allocator. It organizes memory into "heaps" or "heap zones." Its architecture was not designed to support address space mapping, protection, or interrupt time allocation.

The new Memory Manager uses the existing API that is documented in *Inside Macintosh.* If you code to the Macintosh API, you will inherit benefits of the new high-performance Memory Manager.

Things Not to Do

Here is a list of no-nos that may have appeared to work in the old Memory Manager, but may not with the new Memory Manager.

Don't Dispose Blocks Twice

Never dispose of a memory block twice, and don't attempt to operate on a disposed memory

block. Although many traps return `memWZErr` (block is free), it was never a reliable feature and it will be less reliable in the future. When a client disposes of a memory block, the Memory Manager immediately takes control of its data. This is true for both relocatable and nonrelocatable blocks.

There are many subtle ways that a memory block can be disposed of twice. Consider the following example. Joe Ruthless Programmer calls `GetPicture` to display a picture. When he is done he calls `KillPicture` to get rid of it. However, since Joe did not read *Inside Macintosh* carefully, the '`PICT`' resource is still left in the resource map. When the application quits, the System calls `CloseResFile` on the file, which in turn disposes of the resource (again). Pow! You have a handle that was disposed of twice.

Stay Out of Free Blocks

Do not modify data that exists in a free block, and don't attempt to read data or execute code out of a disposed memory block.

Don't Use Fake Handles or Fake Pointers

Call Memory Manager routines only on memory blocks that were created by the Memory Manager. Any call that operates on a pointer block must operate on a pointer block created by the Memory Manager. Do not call `DisposePtr` on data that is on your stack or in your global space. Similarly, any call that takes a handle as a parameter must be a valid handle that was recreated by the Memory Manager. Do not manually create a pointer to a pointer and pass it to the Memory Manager as a handle. It will fail catastrophically—or worse, it will fail subtly.

Don't Make Calls at Interrupt Time

Since all calls either move memory or set low memory global variables (except `BlockMove`), don't even think of calling them at interrupt time or asychronously. In particular, do not call `StackSpace` at interrupt time. The Memory Manager is not an interrupt time storage allocator. Don't use it as such. Doing so will cause many subtle bugs that are difficult to track down.

Don't Touch the Data Structures

Do not access or modify Memory Manager data structures directly. This includes relocatable and nonrelocatable block headers, the heap header, and unused master pointers. If there is an API available, use it. For example, there is an API for locking and unlocking a handle. If you don't use it, you will break in the future. Similarly, don't touch low memory global variables that are maintained by the Memory Manager. Use `SetApplLimit` and `SetGrowZone`.

Keep Your Caches Coherent

Be careful with processors that have dual instruction and data caches, such as the Motorola 68040. Since many clients manipulate code as data, the old Memory Manager flushes caches frequently for compatibility. The new Memory Manager flushes caches only when it moves blocks around. When in doubt, call `FlushInstructionCache` just to make sure.

New Rules to Live By

The new Memory Manager uses larger size block headers and a larger zone header. Clients must leave some slop space for future versions of the Memory Manager. When calculating how large to make a heap, use the following sizes:

Overhead for heap header and trailer	256 bytes
Overhead for each Memory Manager block	32 bytes

Don't Assume Any Stack/Heap Relationships

Don't assume that memory blocks cannot be above your stack. In the future, your stack may be some other place than it is now. However, do call `SetApplLimit` followed by `MaxApplZone` as early as possible in your program to extend your heap to its maximum size.

Create Dynamic Heaps Inside Other Memory Manager Memory Blocks

To create subheaps, allocate a Memory Manager block and call `InitZone` on the beginning of it. When you are done with the subheap, call the Memory Manager to dispose of the block that contains the heap. Do not create heaps on the stack.

Things It's Still OK to Do

Use MoveHHi

It is recommended that you call `MoveHHi` to prevent fragmentation of your heap. For handles created in the system heap you should still call `MoveHHi`, even though this call does nothing today.

Use Existing Tools to Your Advantage

Most popular debuggers for the Macintosh have a heap scramble feature. Use it to debug your code. This feature will ensure that subtle bugs are caught before shipping products to your customers. Similarly, use Zap Handles, Even Better Bus Error, and Double Trouble to ensure compatibility in the future. Although these utilities might not work with the new Memory Manager, you can test with them today using the old one.

Conclusion

With the new high-performance Memory Manager, your applications will run faster. However, you need to follow the rules to be compatible in the future.

Further Reference:

- *Inside Macintosh: Memory*
- Macintosh Technical Note: "OV 04 - Compatibility Why and How"

- Macintosh Technical Note: “OV 11 - The Joy of Being 32-Bit Clean”
- Macintosh Technical Note: “HW 06 - Cache as Cache Can”