The “Why is the Mac IIci Memory Manager so slow?” Bug

Executive Summary:

One way or another, you may have heard an unsettling story: that the Memory Manager on the Mac IIci appears to be slower than that on previous machines.
Sad to say, this is definitely true. The problem afflicts all recent machines, including the Mac IIci, Mac IIfx, Mac IIsi, and Mac LC. These machines are equipped with ROMs that support 32-bit mode under 7.0. This memo aims to describe the symptoms of the problem, its effects on developers, a solution, and what can be done about it.

Basically, the problem involves a new routine that was added when the Memory Manager was made 32-bit clean. This new routine, called MakeCBkF, was added to help optimize the Memory Manager and clean up the application’s memory heap. Unfortunately, MakeCBkF unbalances the Memory Manager such that it is reduced to galacial speeds at times. MakeCBkF is tricked into corrupting a hint that the Memory Manager uses to find free space in the heap. With that hint no longer valid, the Memory Manager’s speed is reduced by a factor of 3 to 10 (yes, it’s 3 to 10 times SLOWER than it should be).

The problem is an ugly one to fix, but it can be done. It’s ugly because the Memory Manager is not vectored (meaning that it’s low-level routines cannot be patched) and because it is mostly self-contained (meaning that we can’t patch a trap that the Memory Manager calls in an attempt to head off the bug). I believe that the only way the bug can be fixed is by replacing the entire Memory Manager with a new one in RAM. Fortunately, this is pretty painless, and involves only a little room from the System heap. DTS currently has a 9K INIT that replaces both the 24-bit and 32-bit Memory Managers. I feel that we could cut this down to 5K by creating 2 INITs: one for the 24-bit Memory Manager
and one for the 32-bit Memory Manager.

Comment from developer:

“Jim Laskey says "I have a whole new machine"! The newest MMInit you just sent has a dramatic impact on performance!!!! Thanks for all your work, well beyond the call of duty. We'll send you a copy of Prograph (the bug fix version we're working on now) (send me your mailing address).

We would like to be able to ship this init with Prograph 2.01, the bug fix version. I suppose you may also want to consider posting it on the various nets and making it available until a new release of the system includes the fixes...?

Again, thank you very much.

Mark Szpakowski (for Jim Laskey)
TGS Systems”

The Technical Details:

In DTS, we’ve been receiving a lot of comments from developers saying they’ve noticed that memory management on the Mac IIci is noticably slower than that on its lesser brethren, such as the Mac IIcx. This can be shown with benchmark programs such as the following:

```
==========================================
Sample Code Listing #1 - Testing NewPtr
==========================================

#define BLOCK_SIZE 200L
#define NUMBER_OF_ITERATIONS 2000

t1 = TickCount();
for(looper=0; looper<NUMBER_OF_ITERATIONS; ++looper)
    result = NewPtr(BLOCK_SIZE);
t2 = TickCount();
result = NewPtr(400000L);
t3 = TickCount();

time = TickCount();   /* record time */


==========================================
Sample Code Listing #2 - Testing NewHandle
==========================================

#define TIMES  10000
```

```
   for( loop = 0; loop < TIMES; loop++ )   /* allocate TIMES handles */
   handle = NewHandle( 1 );

   time = TickCount() - time;/* calculate time difference */
```

The following is a chart that shows what happens when we run the NewPtr program on several different Macintoshes (tests performed under System 6.0.7 from floppy, no MultiFinder, on 8 Meg systems):

```
         |  Ticks
---------+-------
Mac IIfx |    727
Mac IIci |   1386
Mac IIsi |   1423
Mac LC   |   2350
Mac IIcx |    680 !!!
```

Similarly, Test #2 took 1080 ticks to execute on a Mac IIfx, but only 250 ticks on a Mac IIcx. Adding enough calls to MoreMasters() to deal with 10,000 handles reduced these times to 90 and 75 ticks respectively. Taking into account the speed differences between the IIcx and IIfx, this means that the Mac IIfx runs the test 3-10 times slower than the Mac IIcx.

I’ve seen some people throw up their hands and say “Hey! That’s life! With the extra overhead of deciding whether to call the 24-bit Memory Manager or 32-bit Memory Manager, the IIci is actually slower. There’s nothing that can be done about it.”

Fortunately, this turns out NOT to be the case. I didn’t think that the overhead of an extra vector jump or two was going to slow down the Memory Manager by a factor of up to 10. So with the above test programs, the perfomance tools, and the ROM sources, I tracked down the problem.

It seems that it all hinges on a new routine called a24MakeCBkF (and a32MakeCBkF in 32-bit mode). Not only is there a bug that causes a24MakeCBkF to work incorrectly, but even if it DID work correctly, it adversely affects the Memory Manager in two other spots.

As it looks today, here is a24MakeCBkF:

```
a24MakeCBkF
            Movem.L A1/D0,-(SP)         ;save A1, D0
            Move.L  A0,A1               ;get start of free block address
            Add.L   D0,A1               ;get beginning of next block
@tryNext
            Tst.B   TagBC24(A1)         ;is next block free
            Bne.S   @notFree            ;branch if not a free block
            CmpA.L  BkLim(A6),A1        ;is it the last free block
            Bcc.S   @notFree            ;branch, if yes
            Add.L   TagBC24(A1),D0      ;add size of next free block
            AddA.L  TagBC24(A1),A1      ;advance A1 to next block
            Bra.S   @trynext            ;check next block
@notFree
            Move.L  D0,TagBC24(A0)      ;Set tag and byte count.
            Clr.B   TagBC24(A0)         ;Clear tag and offset.
            Move.L  A0,AllocPtr(A6)     ;update allocPtr to point here
            RTS                         ;Return to caller.
```

This routine is used to replace a24MakeBkF. Both of these routines are responsible for marking a block as being free. However, a24MakeCBkF does a little more. Instead of just marking the given block as free, it looks for trailing blocks that might also be free. If it finds any, it combines them into one big block, and marks that block as free. Additionally, it sets AllocPtr (the roving pointer) to point to the new free block.

Before going on, I should probably explain what AllocPtr does. Obviously, the Memory Manager has a constant need to find free space in the heap. This can happen if it needs to create a new block, move a relocatable block, or create space after a block that is being expanded. To help expedite things, the Memory Manager keeps a hint in AllocPtr.
This points to a “good place to find a free block” and almost always is in the vicinity of a bunch of free blocks. This relieves the Memory Manager from having to do a full heap search when it needs some free space.

AllocPtr is used by a routine called BFindS (for BlockFindSpace, or some other nifty name). BFindS takes a look at AllocPtr: if it is NIL, BFindS starts its search from the beginning of the heap, otherwise it starts at the block pointed to by AllocPtr. If it starts searching with

AllocPtr and hits the end of the heap without finding anything, it clears AllocPtr and starts over from the beginning of the heap.

The First Problem (24 bit mode only):

When a handle needs to be moved in the heap, a routine named RelocRel is called. As part of its algorithm, it fetches the master pointer for the block. This pointer is subsequently passed to some other helper subroutines. In the past, one of those subroutines was MakeBkF, which was benign. Now that MakeCBkF is called instead of MakeBkF, AllocPtr gets set to the value to that master pointer. The problem is that the master pointer bits, if any, ARE STILL SET! This results in a dirty pointer being stored in AllocPtr. When BFindS attempts to use AllocPtr as a hint, it immediately determines that the pointer is past the end of the heap, and that it should start over. In reality, if the upper byte had been stripped off before the comparison was made, everything would have been OK. As it is, the hint is rendered useless, and the Memory Manager starts its free space searches from the beginning of the heap every single time.

The solution to this is to modify a24RelocRel, which is where the dirty pointer is generated. Find the line that says:

```
Move.L  (A2),A0          ;Points to source block
```

and replace it with:

```
Move.L  (A2),D0          ;Points to source block
And.L   Lo3Bytes,D0        ;don't store the naughty bits
Move.L  D0,A0
```

This results in the correct value being stored in AllocPtr. However, two other problems arise now that AllocPtr is being changed in an algorithm where it wasn’t being set
previously.

The Second Problem (24 & 32 bit mode):

When the Memory Manager is called upon to create a new non-relocatable block, it tries to create the block as low in memory as possible. It does this by looking for a range of the heap that is filled

with nothing but free and/or relocatable blocks in sufficient quantity. When it finds one, it starts coalescing the blocks together. Free blocks it just annexes. When it gets to a relocatable block, it moves the block high in the heap, and then marks the old spot as free by calling MakeCBkF. This has the effect of unconditionally setting AllocPtr to that newly freed block. However, this does us no good, as that block is immediately merged into the big block it is creating for the NewPtr call. When that happens, the Memory Manager checks to see if the block it's annexing is referenced by AllocPtr. If so, it invalidates AllocPtr by setting it to NIL. Since we just called MakeCBkF, we guaranteed that the block that's going away is referenced by AllocPtr, and we immediately lose the hint we set up for ourselves.

For a similar reason, the unconditional setting of AllocPtr affects NewHandle. Take the case above where we create 10,000 handles without creating master pointer blocks first.
The Memory Manager starts out OK, and allocates 60 or so handles before it needs to create another master pointer block. Since it needs to allocate this low in memory, it must first move the handles out of the way. So it moves one block up and frees up its old space, setting AllocPtr to point to that new free space. This process is repeated as the program runs, such that eventually AllocPtr is pointing to a free block at the bottom of 8000 or 9000 handles. Everytime it needs to create a new handle, it must climb this stack before it can find free space. On the other hand, when MakeBkF was called in the older ROMs, AllocPtr was kept pointing high in the heap where it was most useful.

Therefore, it seems that pointing AllocPtr to the newly created free block is not the right thing to do. But why was it done? My guess is that it was to take into account the possibility that AllocPtr could be pointing any of the free blocks we are adding to the one we marking as free. If that happened, AllocPtr would be pointing into the MIDDLE of a block, which would cause problems later.

The solution, then, is to change AllocPtr only if it points to a block that is being added to the end of the one in front of it. This can be done with the following version
of MakeCBkF:

```
a24MakeCBkF
            Movem.L A1/D0,-(SP)         ;save A1, D0
```

```
            Move.L  A0,A1               ;get start of free block address
            Add.L   D0,A1               ;get beginning of next block
@tryNext
            Tst.B   TagBC24(A1)         ;is next block free
            Bne.S   @notFree            ;branch if not a free block
            CmpA.L  BkLim(A6),A1        ;is it the last free block
            Bcc.S   @notFree            ;branch, if yes

            Cmp.L   AllocPtr(A6),A1     ; kaar 90.11.14
            Bne.S   @1                  ; kaar 90.11.14
            Move.L  A0,AllocPtr(A6)     ; kaar 90.11.14

@1
            Add.L   TagBC24(A1),D0      ;add size of next free block
            AddA.L  TagBC24(A1),A1      ;advance A1 to next block
            Bra.S   @trynext            ;check next block
@notFree
            Move.L  D0,TagBC24(A0)      ;Set tag and byte count.
            Clr.B   TagBC24(A0)         ;Clear tag and offset.
;            Move.L  A0,AllocPtr(A6)    ;update allocPtr to here  <kaar>
            Movem.L (SP)+,A1/D0         ;restore A1 ,D0
            RTS                         ;Return to caller.
```

As you can see, we remove the unconditional setting of AllocPtr, and add a check to see if it currently points to a block that's about to go away. If so, only then do we point it to the block that is being freed.

With these two changes, our before and after benchmark chart looks likes this:

	Before	After
Mac IIfx	727	315
Mac IIci	1386	626
Mac IIsi	1423	663
Mac LC	2350	1043
Mac IIcx	680	

Additionally, the NewHandle test drops to 90 ticks without calling MoreMasters, and even lower to 29 ticks if we DO call MoreMasters.

Other optimizations (24 & 32 bit mode):

During the study of these problems and solutions, it became clear to me that we gained nothing by invalidating AllocPtr by setting it to NIL. This is done in two places in the Memory Manager, both of which could be made smarter.

The first is in MakeSpace. During the process of combining free blocks and newly vacated blocks into one big block, the Memory Manager checks if the block being annexed is referenced by AllocPtr. If it is, AllocPtr is invalidated by setting it to NIL. I think that a much better solution would be to point AllocPtr to the next block in the heap
instead, thus:

```
@MSpFree
            Move.L  A3,A0               ;Address of block being released.
            Move.L  TagBC24(A0),D0          ;Size of block
            Cmp.L   AllocPtr(A6),A0         ;Does the rover point here?
            BNE.S   @MSpNext                ;Skip if not . . .
;            Clr.L   AllocPtr(A6)           ; kaar 90.11.14 nuked this
            And.L   Lo3Bytes,D0               ; kaar 90.11.12
            Add.L   D0,AllocPtr(A6)           ; kaar 90.11.12
```

But just to be vicious about it, MakeSpace takes another stab at invalidating AllocPtr as it exits. It does this with a good reason: a free space has just been created, and BFindS is about to be called to turn it into a new block. However, BFindS starts searching from AllocPtr, which could be almost anywhere. Therefore, AllocPtr is cleared so that BFindS will start from the beginning of the heap and have a better chance of finding the block we created more quickly.

A better choice would be to just set AllocPtr to the block we freed up so that BFindS will find it immediately, like this:

```
@MSpExit
;            Clr.L   AllocPtr(A6)           ; kaar 90.11.14 nuked this
            Move.L  A2,AllocPtr(A6)         ; kaar 90.11.14
```

This is good, but reveals a bug in SetSize. SetSize is also in the business of moving relocatable blocks out of the way so that it can annex the free space to the block we want to expand. It does this by calling MakeSpace, which exits after setting AllocPtr to the new free

space. SetSize then annexes this free space to the block we are growing, but it doesn’t check to see if the block going away was referenced by AllocPtr. We therefore add a check for this in SetSize, setting AllocPtr to the block being grown if so:

```
@SSEnough                               ;Enough room has been found.
            BSR.S   ClearGZStuff         ;mark gz stuff as done

            Cmp.L   AllocPtr(A6),A2       ; kaar 90.11.14
            Bne.S   @11                 ; kaar 90.11.14
            Move.L  A0,AllocPtr(A6)       ; kaar 90.11.14
@11
```

With these optimizations, the time of 315 ticks for running the NewPtr test on a Mac IIfx decreases to 175 ticks.