New Technical Notes

Macintosh ®

Developer Support

HW 6 - Cache As Cache Can

Hardware

Revised by: Rich Kubota, Craig Prouse April 1992
Written by: Andrew Shebanow October 1989

This Technical Note documents cache behavior, manipulation of processor caches, and manipulation of external caches on Macintosh models that incorporate these features. It also describes how system software uses a memory management unit (when available) to implement special caching options.

Changes since October 1991: Described use of AppleTalk Transition Queue event, ATTransSpeedChange, when altering the 68040 cache state on the fly. This call must be issued so that LocalTalk can reevaluate its timers. Otherwise LocalTalk becomes disabled.

Cache Machines

The Motorola MC68020 microprocessor includes a 256-byte internal instruction cache. The MC68030 includes a similar-size instruction cache plus a 256-byte writethrough data cache. The MC68040 has much larger caches, 4K of instructions, and 4K of data. It also supports copyback caching in addition to the writethrough caching used by the MC68030.

The difference between writethrough and copyback caching is a matter of whether data writes go directly and immediately to main memory, or whether they go only as far as the data cache to be copied back to main memory later (if necessary) in a highly optimized fashion.

The MC68030 and MC68040 include memory management units internally. Besides the ability to divide memory into logical pages and provide memory access control, these memory management units can also associate cachability attributes with individual pages of memory, affecting how data is cached on a page-by-page basis.

Stale Data (Baked Fresh Daily)

Caching greatly improves overall system performance but introduces the problem of stale data, or inconsistency between cached data and the data in actual memory (RAM). In certain cases, cache maintenance instructions are necessary to maintain coherency between cache and main memory.

Stale Instructions

The first time when stale data becomes a problem occurs when writing self-modifying code on the MC68020 or any other processor with an instruction cache. The instruction cache

remembers, separately from main memory, many of the instructions it has recently executed. If the processor executes an instruction, later changes that instruction in memory, and then tries to execute the new instruction at the same address, there is a probability that the original instruction is still cached. Since the cache is used before main memory, the old instruction may be executed instead of the new one, resulting in incorrect program operation.

To prevent this, any time a program changes an executable instruction in memory, it must flush the instruction cache before attempting to execute the modified instruction. Flushing a cache invalidates its entries, forcing the processor to refill cache entries from main memory. In part, this defeats the purpose of a cache and hurts performance. Nevertheless, the alternative is incorrect program operation. This serves to emphasize that caches must always be flushed judiciously in order to maintain correct operation and optimal performance.

As described, self-modifying code is not just code that changes itself directly as it executes. It can be much more subtle. Code that modifies a jump table entry is modifying executable code and must flush the instruction cache. Patch installation code often copies code from one block of memory into another and may modify one or more `JMP` instruction operands in order to get back to the original routine—either technique requires flushing the instruction cache.

Stale Data

With the addition of the data cache in the MC68030, performance is further enhanced, but another cache offers another source of stale data.

Let's say that you have a whizzy disk controller card that supports DMA. The board reads command buffers from the main CPU's memory area and writes status information back to the command buffer when done. Before the command is started, the MC68030 sets up the command buffer and zeroes the status code (the following figures are not to scale).

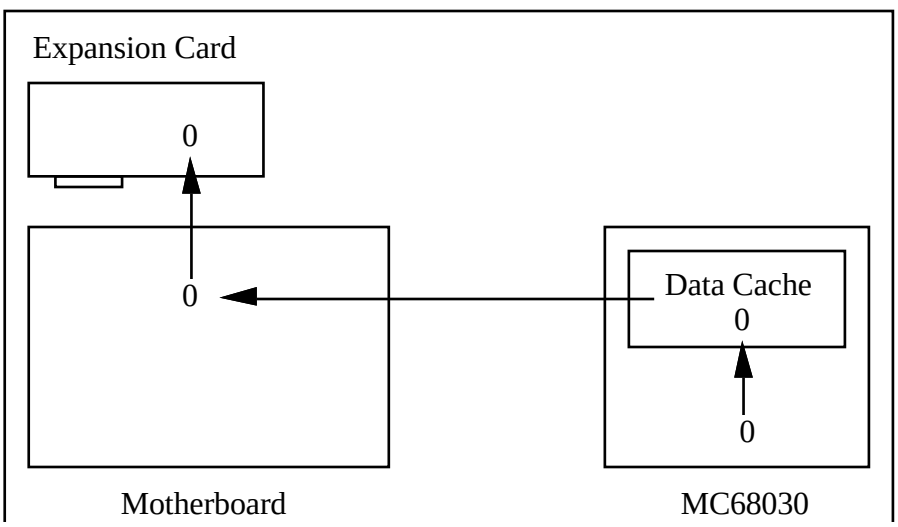

Figure 1 Write (Writethrough Cache)

At this point the cache and the memory both contain the value 0, since the MC68030's cache is writethrough (that is, it always writes data to memory immediately). Now the MC68030 starts the command running and waits for an interrupt from the disk controller card. It then reads back the status from the command buffer, which is modified by the DMA card.

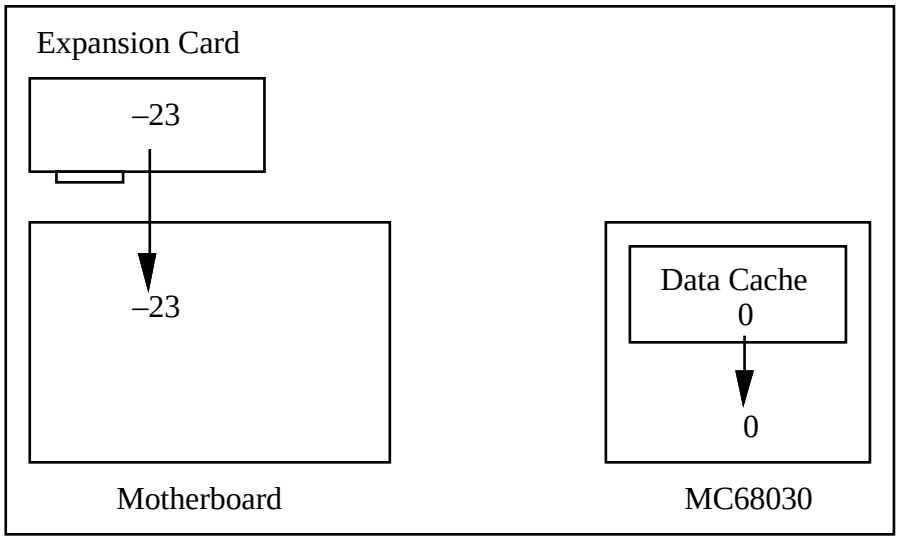

Figure 2 Read (From Cache)

Oops! Because the status code's value is already cached, the MC68030 thinks that the status is 0, even though the actual value in memory is –23. This type of thing can cause some very hard-to-find bugs in your driver.

Copyback Data and Stale Instructions

There is another type of cache called a copyback cache that is supported by more advanced microprocessors like the MC68040. A copyback cache further improves system performance by writing data to external memory only when necessary to make room in the cache for more recent accesses, or when explicitly "pushed" by system software. This is extremely valuable for relatively small, short-lived data that are accessed frequently but don't need to persist for a long time, like local stack frames in C and Pascal function calls.

This increase in performance again comes at some cost in terms of maintaining cache coherency. Here, the problem is twofold. Fundamentally, a datum that is "written to memory" isn't really in memory (meaning main RAM) until it's pushed out of the data cache. When performing DMA, it is necessary to push data cache contents into memory before instructing alternate bus masters to look for it; they'll only find stale data if it's still cached. Second, and perhaps even more important, the instruction and data caches are completely independent of each other. When fetching instructions, the processor looks in only two places: first the instruction cache, then main memory. It does not look in the data cache. When performing the types of operations described above that can cause a stale instruction cache, one must remember that it is impossible to make the instruction cache and memory coherent if memory itself is stale! The data cache must be flushed; then and only then can the instruction cache refill with the valid data the processor has written.

Here, some code writes the _LoadSeg trap to memory as part of a jump table update. Figure 3 indicates what happens if only the instruction cache is flushed. When execution later proceeds through that jump table entry, the processor fetches the opcode from that address and gets zonked with an illegal F-line exception. Why? _LoadSeg is still in the data cache. The code responsible for maintaining the jump table failed to push the contents of the data cache before invalidating the instruction cache. This certainly causes problems on the MC68040.

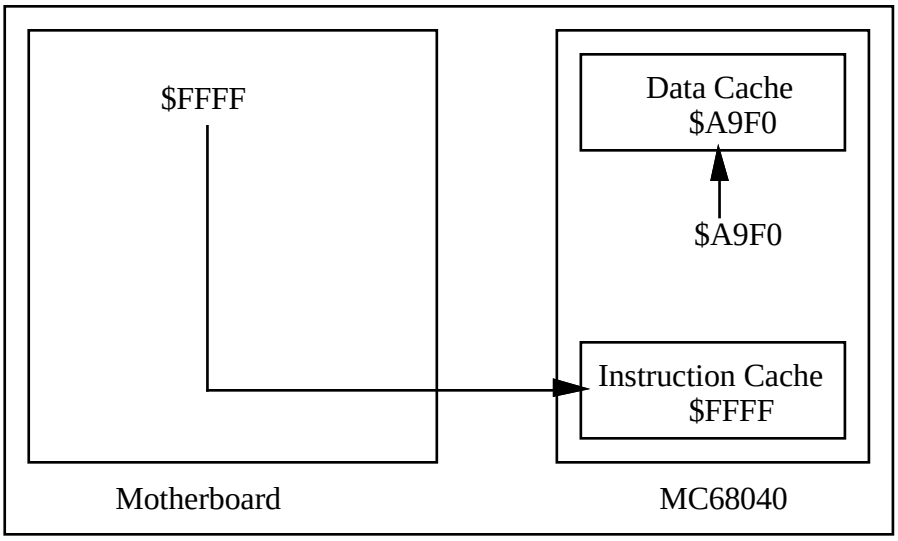

Figure 3 Write (Copyback Cache) and Fetch

Another similar problem applies to the time at which cache flushing is performed. When using a writethrough data cache, it is acceptable to invalidate the instruction cache first and then modify instructions in memory. With a copyback data cache, it is imperative to make changes to memory first and then flush caches. Again, this ensures that copyback data is written to memory before the instruction cache attempts to refill from memory. The key point to remember is that the MC68040 instruction cache always reads from memory, never from the data cache.

Figure 4 shows the path that an instruction properly takes when it is first written as data by a program that modifies instructions in memory.

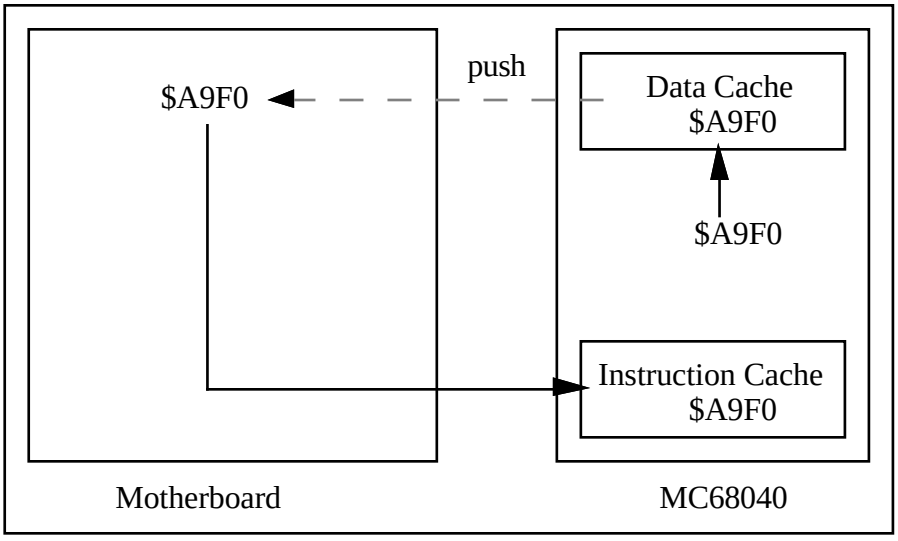

Figure 4 Write (Copyback Cache), Push, and Fetch

It's worth noting here that although pushing copyback data to memory and invalidating (flushing) the cache are conceptually different operations, they are at least for the MC68040 irrevocably connected. This makes flushing the data cache for the sake of pushing its contents to memory a potentially expensive one. Valid cache data is essentially lost when it is pushed and must be read from main memory if it is to be accessed again. This should be another reinforcement that cache flushing must be performed judiciously. It is possible to flush only a portion of the MC68040 caches, and software that flushes caches frequently should consider this optimization to avoid unnecessary performance degradation when running on this processor. See the interfaces provided below.

What Is Apple's Part in This?

There are two answers to this question. First, there are things that Apple has done in ROM to make life easier while dealing with a caching processor. Second, there are functions provided in ROM or in system software to allow developers to take some control of their own destinies.

Things That Happen for You

Ever since the Macintosh II made its debut, it has been flushing the instruction cache. It does so at a number of critical points where code may be moved to a new location, potentially leaving memory and the instruction cache incoherent. Specifically, there are a number of traps that have the potential to move code around memory. In each of these cases, the instruction cache is flushed by system software or ROM.

```
_BlockMove  _LoadSeg
_Read _UnloadSeg
```

Warning: The `_BlockMove` trap is not guaranteed to flush caches for block sizes 12 bytes or less. This is a performance optimization. `_BlockMove` is called often by system software to move small blocks of memory that are not executable instructions. Flushing the cache in all such cases causes significant performance degradation. When moving very small blocks of code with _BlockMove, use one of the explicit cache flushing routines described below.

Note: C programmers should not assume that the standard library function memcpy() invokes _BlockMove. An explicit cache flush is required after moving code with memcpy().

In general, there may be others. As a rule of thumb, the instruction cache needs to be flushed explicitly only as a result of actions taken by user code, not as the result of anything a trap might have done. Traps can take care of themselves.

A memory management unit allows individual pages of memory to be marked noncachable. In current Macintosh implementations, NuBus™ and I/O address spaces are always marked noncachable—the processor won't cache memory stored at NuBus or I/O addresses. This solves any problems of stale data when processor/DMA "mailboxes" are located in NuBus memory and eliminates the fundamental problem of stale data at memory-mapped I/O locations. Data at RAM and ROM addresses are cachable, which makes sense and maximizes performance.

Since DMA still poses a problem when common buffers are located in main RAM, it would seem that there should be greater intrinsic support for specifying cachability. There is. In order

for DMA masters to be compatible with abstract memory architectures like those defined by the Macintosh IIci and even more so by virtual memory, they must use the GetPhysical routine. Before using GetPhysical, a range must always be locked with LockMemory. Since this sequence is so commonly required when performing DMA, the LockMemory routine has the effect of either disabling the data cache or marking the corresponding pages noncachable, depending on what's possible and what makes the most sense. In many cases, therefore, it is unnecessary to explicitly flush the data cache. If common DMA buffers are locked with LockMemory, the operating system ensures cache coherency at least for those buffers.

To ensure compatibility with existing code while taking advantage of copyback cache mode, the FlushInstructionCache function on an MC68040 actually flushes both caches using the CPUSHA BC instruction. This prevents the need for modification of correct existing code which properly flushes the instruction cache with FlushInstructionCache. If code is written properly for the MC68020 and MC68030, it will work on the MC68040 as well, without modification. If code is written incorrectly or directly manipulates the CACR register of these processors it will fail on the MC68040. When modifying code in memory or moving code about memory, use FlushInstructionCache before executing that code.

Facilities That Are Provided for You

Apple provides some system calls that let you flush the data and instruction caches without using privileged instructions (which is, as you should all know by now, a major no-no).

Following are the interfaces for these calls, for MPW Pascal and C (respectively):

```
FUNCTION SwapInstructionCache (cacheEnable: BOOLEAN) : BOOLEAN;
pascal Boolean SwapInstructionCache (Boolean cacheEnable);
```

This call enables or disables the instruction cache according to the state passed in `cacheEnable` and returns the previous state of the instruction cache as a result.

```
PROCEDURE FlushInstructionCache;
pascal void FlushInstructionCache (void);
```

This call flushes the current contents of the instruction cache. This has an adverse effect on CPU performance, so only call it when absolutely necessary.

```
FUNCTION SwapDataCache (cacheEnable: BOOLEAN) : BOOLEAN;
pascal Boolean SwapDataCache (Boolean cacheEnable);
```

This call enables or disables the data cache according to the state passed in `cacheEnable` and returns the previous state of the data cache as a result.

```
PROCEDURE FlushDataCache;
pascal void FlushDataCache (void);
```

This call flushes the current contents of the data cache. This has an adverse effect on CPU performance, so only call it when absolutely necessary.

Note: Before you call any of these routines, make sure that the _HwPriv ($A198) trap is implemented, or your program will crash. _HwPriv is implemented in the Macintosh IIx ROMs and later, as well as System 6.0.3 and later. The correct way to check for the trap is using the TrapAvailable function documented in *Inside Macintosh* Volume VI (pages 3-7 to 3-9).

These calls are provided as part of the MPW 3.1 library. For those of you without MPW 3.1 or later, you can use the following MPW assembly-language glue:

```
            CASE OFF

_HwPriv     OPWORD  $A198

SwapInstructionCache PROC EXPORT
            MOVEA.L (A7)+,A1        ; save return address
            MOVEQ   #0,D0           ; clear D0 before we shove Boolean into it
            MOVE.B  (A7)+,D0        ; D0 <- new mode
            MOVE.L  D0,A0           ; _HwPriv wants mode in A0
            CLR.W   D0              ; set low word to 0 (routine selector)
            _HwPriv
            MOVE.W  A0,D0           ; move old state of cache to D0
            TST.W   D0              ; if nonzero, cache was enabled
            BEQ.S   WasFalse        ; if zero, leave result false
            MOVEQ   #1,D0           ; set result to true
WasFalse:
            MOVE.B  D0,(A7)         ; save result on stack
            JMP     (A1)
            ENDPROC

FlushInstructionCache PROC EXPORT
            MOVEA.L (A7)+,A1        ; save return address
            MOVEQ   #1,D0           ; set low word to 1 (routine selector)
            _HwPriv
            JMP     (A1)
            ENDPROC

SwapDataCache PROC EXPORT
            MOVEA.L (A7)+,A1        ; save return address
            MOVEQ   #0,D0           ; clear D0 before we shove Boolean into it
            MOVE.B  (A7)+,D0        ; D0 <- new mode
            MOVE.L  D0,A0           ; _HwPriv wants mode in A0
            MOVE.W  #2,D0           ; set low word to 2 (routine selector)
            _HwPriv
            MOVE.W  A0,D0           ; move old state of cache to D0
            TST.W   D0              ; if nonzero, cache was enabled
            BEQ.S   WasFalse        ; if zero, leave result false
            MOVEQ   #1,D0           ; set result to true
WasFalse:
            MOVE.B  D0,(A7)         ; save result on stack
            JMP     (A1)
            ENDPROC

FlushDataCache PROC EXPORT
            MOVEA.L (A7)+,A1        ; save return address
            MOVEQ   #$3,D0          ; set low word to 3 (routine selector)
            _HwPriv
            JMP     (A1)
            ENDPROC
```

There are two additional calls whose interfaces follow. Each requires a little explanation.

The first call is `FlushCodeCache`, which simply invokes the `_CacheFlush` ($A0BD) trap. This trap's function is to make the instruction cache coherent with memory. On the MC68020 and MC68030 it simply flushes the instruction cache. On the MC68040 it also flushes the data cache for copyback compatibility. The advantage of `FlushCodeCache` as opposed to

FlushInstructionCache is that it was implemented before the _HwPriv trap, and thus can be used on the Macintosh II while running older system software.

In general, FlushInstructionCache is still the preferred application-level cache flushing mechanism. FlushInstructionCache calls FlushCodeCache and is therefore a higher-level call conceptually. FlushCodeCache may be useful where FlushInstructionCache proves unsuitable, or as an alternative to the next call, FlushCodeCacheRange. Obviously, before calling FlushCodeCache, be certain that _CacheFlush is implemented.

Note: If the processor has a cache to flush, this trap should be properly implemented, because ROM and system software use this trap's vector to do their own cache flushing. In fact, FlushInstructionCache itself uses this vector. This should be of particular interest to accelerator card developers.

```
            MACRO
            _FlushCodeCache
            _CacheFlush
            ENDM

PROCEDURE FlushCodeCache;
    INLINE $A0BD;

void FlushCodeCache (void) = 0xA0BD;
```

The second call is FlushCodeCacheRange. FlushCodeCacheRange is an optimization of FlushCodeCache designed for processors like the MC68040 which support flushing only a portion of the cache. (The MC68020 and MC68030 do not support this feature and FlushCodeCacheRange simply flushes the entire instruction cache on those processors.) As described earlier, pushing and flushing cache entries are linked and flushing the entire cache after a small change like a jump table entry can be expensive. FlushCodeCacheRange allows one to request that only a specific memory range be flushed, leaving the rest of the cache intact. Note that this is only a request and that more than the requested range may be flushed if it proves inefficient to satisfy the request exactly. Also, FlushCodeCacheRange may not be implemented for some older versions of system software that are not MC68040-aware. If not, FlushCodeCacheRange returns hwParamErr (–502) and it is necessary to flush the entire cache instead, probably using FlushCodeCache. If FlushCodeCacheRange succeeds it returns noErr (0). Before calling FlushCodeCacheRange, be certain that _HwPriv is implemented.

```
; _FlushCodeCacheRange takes/returns the following parameters:
;    -> A0.L =   Base of range to flush
;    -> A1.L =   Length of range to flush
;    <- D0.W =   Result code (noErr = 0, hwParamErr = -502)
            MACRO
            _FlushCodeCacheRange
            moveq   #9,d0
            _HwPriv
            ENDM

FUNCTION FlushCodeCacheRange (address: UNIV Ptr; count: LongInt) : OSErr;
    INLINE $225F,   { MOVEA.L (SP)+,A1 }
           $205F,   { MOVEA.L (SP)+,A0 }
           $7009,   { MOVEQ   #9,D0    }
           $A198,   { _HwPriv          }
           $3E80;   { MOVE.W  D0,(SP)  }

// MPW C 3.2 makes register-based inline calls very efficient.
#pragma parameter __D0 FlushCodeCacheRange(__A0,__A1)
```

```
OSErr FlushCodeCacheRange (void *address, unsigned long count) =
    {0x7009, 0xA198};

/* MPW C 3.1 and earlier, and THINK C™ should declare the function as   */
/* "pascal" and use the same inline constants as the Pascal interface: */
pascal OSErr FlushCodeCacheRange (void *address, unsigned long count) =
    {0x225F, 0x205F, 0x7009, 0xA198, 0x3E80};
```

Caching Consequences—LocalTalk

As noted above, altering the state of the data/code cache significantly affects the performance of the 68040 processor. This change in effective CPU speed may affect any background process that is dependent on the processor speed remaining constant. LocalTalk is an example of one such affected process, as it employs speed sensitive timing loops. The change in CPU speed affects the LocalTalk timers, to the extent that the LocalTalk no longer functions correctly if it is the current AppleTalk connection.

Fortunately, the AppleTalk Transition Queue mechanism can be used to notify LocalTalk of the change in effective CPU speed. Upon notification, LocalTalk recalculates its timer values to match the current CPU speed. Refer to *Inside Macintosh* Volume VI, page 32-17, and to Technical Note NW 13 - "AppleTalk: The Rest of the Story" for additional information on the use of the AppleTalk Transition Queue.

The following code demonstrates the use of the `ATEvent` procedure to send the `ATTransSpeedChange` event. The `ATEvent` call is provided as part of the MPW 3.2 library.

Important Note: Issue the `ATTransSpeedChange` event **only** at `SystemTask` time!

```
USES AppleTalk;                             { ATEvent prototyped in AppleTalk unit,
                                              MPW 3.2 }

CONST
    ATTransSpeedChange = 'sped';            {change in cpu speed transition }

PROCEDURE NotifyLocalTalkSpeedChange;

    BEGIN
        if LAPMgrExists THEN                { check LAP Manager exists, see Tech Note
                                              311 }
                                            { for the code for LAPMgrExists }
            ATEvent(longint(ATTransSpeedChange), NIL);
                                            { notify speed change event }
    END;
```

Note that only LocalTalk drivers that are included with AppleTalk version 57 or greater, respond to the ATTranSpeedChange event. System 7.0.1 for the Quadra's, is supplied with AppleTalk version 56. AppleTalk version 57 is available by using the AppleTalk Remote Access Installation program, or the Network Software Installer version 1.1. Licensing for AppleTalk can be arranged by contacting Apple Software Licensing. Software Licensing can be reached as follows:

Software Licensing
Apple Computer, Inc.
20525 Mariani Avenue, M/S 38-I
Cupertino, CA 95014
MCI: 312-5360
AppleLink: SW.LICENSE
Internet: SW.LICENSE@AppleLink.Apple.com
(408) 974-4667

AppleTalk version 53 or greater is required to handle the ATEvent call, however, nothing bad will happen if you issue the ATTranSpeedChange transition event under AppleTalk versions 53 - 56. It is important to check that the LAP Manager is implemented before issuing the `ATEvent` call. See Tech Note 311 for a description of the `LAPMgrExists` function.

External Caches

The Macintosh IIci and Macintosh IIsi support external cache cards. Because of the way these caches work, cache coherency is not much of a problem. In fact these caches are usually enabled full-time and their operations are totally transparent to all well-behaved hardware and software. Still, there are corresponding cache control functions to enable, disable, and flush these cache cards. If _HwPriv is implemented, the following routines may be used:

```
            MACRO
            _EnableExtCache
            moveq   #4,d0
            _HwPriv
            ENDM

PROCEDURE EnableExtCache;
    INLINE $7004,$A198;

void EnableExtCache (void) = {0x7004, 0xA198};


            MACRO
            _DisableExtCache
            moveq   #5,d0
            _HwPriv
            ENDM

PROCEDURE DisableExtCache;
    INLINE $7005,$A198;

void DisableExtCache (void) = {0x7005, 0xA198};


            MACRO
            _FlushExtCache
            moveq   #6,d0
            _HwPriv
            ENDM

PROCEDURE FlushExtCache;
    INLINE $7006,$A198;

void FlushExtCache (void) = {0x7006, 0xA198};
```

Further Reference:

- *Inside Macintosh*, Volume V, Operating System Utilities
- *Inside Macintosh*, Volume VI, Compatibility Guidelines
- *Designing Cards and Drivers for the Macintosh Family*
- *M68000 Family Programmer's Reference Manual*
- *M68020 32-Bit Microprocessor User's Manual*
- *M68030 Enhanced 32-Bit Microprocessor User's Manual*
- *M68040 32-Bit Third-Generation Microprocessor User's Manual*