OV 13 - 10+ Commandments

Overview

Written by: Rich Collyer and Dave Radcliffe June 1993

The world of Macintosh is changing rapidly. With a plethora of new Macintosh CPUs and the prospect of even stranger, non-68K CPUs on the horizon, how's a programmer supposed to keep up? Maybe it's time to review some basic tenets of Macintosh programming and point out some future pitfalls. Not only is it important that Apple evolve new hardware; Apple must evolve the operating system as well. The issues discussed here affect the ability of Apple to transform the Macintosh Operating System into a modern operating system. As you write new code, or review old code, be aware of these issues. If you can't deal with them immediately, at least flag problem areas with appropriate comments so you can fix them in the future. By doing so, you'll help Apple bring you those modern operating system features you've been screaming for that much sooner.

Topics

- Rules to compatible Macintosh programming
- To live or die in the Macintosh Operating System
- How to make your application run into the late nineties

1. Write in ANSI C or C++

This is a bit of a religious issue. There are some people who I have heard say that they will be dead and cold before their assembler can be taken from their fist. Unfortunately, assembly code is very hard to port to new CPUs and Pascal is falling out of favor with the people inside of Apple who make the Macintosh compilers and write system software. It is very likely that Pascal will continue to be supported by third-party developers, but the Macintosh Operating System is slowly being converted to C and C++. In making this conversion, the special features of C are being used. As a result, Pascal programmers and compiler writers will need to think in C and make the appropriate conversions of the data and function calls to connect the C conventions with those of Pascal.

Religious or not, the Macintosh Operating System is being written in C and C++ and to make sure that your code is more compatible with our system, we recommend that you learn to love C. If you make the investment now you are assured the easiest transition possible to new platforms. Besides, it is also very hard to make hand-tuned assembly that is better than code produced by a good optimizing C compiler, especially on RISC-type machines.

You should take full advantage of the features ANSI C provides. For example, you should turn on compiler options to require prototypes and make sure that all your own functions have prototypes. Be aware that use of "old style" function definitions in MPW C will defeat prototype checking. For example, the following definition of `DoEvent` allows any 4-byte parameter to be passed as the `evtPtr` parameter:

```
void DoEvent (evtPtr)
EventRecord *evtPtr;
{
       .
       .
       .
}
```

To get the most out of prototyping, you must always use “new style” function declarations and definitions:

```
void DoEvent (EventRecord *evtPtr)
{
       .
       .
       .
}
```

Do not assume the C compiler understands Pascal calling conventions. In particular, do not assume the C compiler will automatically pass toolbox structures larger than 4 bytes by reference; do it yourself.

Never use the type `int.` Purists may argue that proper use of `int` gives you the most portable code. But the Macintosh Toolbox is pretty rigid in its use of 16-bit and 32-bit values and experience shows use of `int` just leads to trouble. Use `short` and `long` instead. If you are uncomfortable with `short` and `long`, create your own `typedefs` such as `int16` and `int32` so you can alter them for different compilers.

Using direct functions in MPW C or inline assembly in Think C is often unavoidable with the current Toolbox, but it is not portable. You should isolate and conditionalize such code. If you write assembly routines for performance reasons, considering writing a C version, for portability, at the same time you write the assembly version.

2. Align Data Structures

The Motorola 68K microprocessors have always been very tolerant of misaligned data structures, but modern, cached computer architectures don’t like having to support misaligned data structures. Chances are that the microprocessors that Apple uses in its CPUs will continue to support misaligned data structures, but you will probably find that applications will run considerably faster if the data structures are aligned. This is already the case in the 68040, and it will become more and more important in the future. So if your structure declarations look something like:

```
struct MatchRec {
    unsigned short red;             // 16 bit variable
    unsigned short green;
    unsigned short blue;
    long matchData;                 // 32 bit variable
};
```

Then you may want to change them to look more like:

```
struct MatchRec {
    long matchData;                  // 32 bit variable
    unsigned short red;              // 16 bit variable
    unsigned short green;
    unsigned short blue;
};
```

The problem with the first example is that the long field `matchData` is not aligned on a 32-bit boundary. The third short field, `blue`, offsets the long field from the 32-bit boundary by 16 bits.

3. Don’t Depend on 68K Run-time Model (Stacks, A5, Segmentation…)

The 68K run-time model contains many features that are extremely machine dependent (such as A5 worlds) or don’t make sense in a modern operating system (such as segmentation). Such features should continue to work in a 68K environment, but when you port code to other platforms, assumptions you make about the run-time environment may no longer be valid.

Beware of the assumptions made in the following areas:

- A5 world. This provides two features: application global data and function references, and access to QuickDraw globals. Similar functionality will be provided in other run-time environments, but the method of access will be different.

- Register conventions. A5 is a specific example of a dependency on register conventions. Other examples can be found, such as depending on return values in D0, or A7 being the stack. Beware of similar dependencies on the 68K register model. It will undoubtedly be different on other platforms.

- Calling conventions. Besides emphasis on C calling conventions, different run-time environments are likely to have idiosyncratic calling conventions. Beware of assumptions based on return values, or parameter ordering, location, size, or alignment.

- Stack structure. Different run-time environments will use the stack in different ways. Don’t assume you know the layout of stack frames, or indeed, even the layout within a stack frame.

- Segmentation. Segmentation will certainly be different, or even nonexistent on future platforms. In most cases this will be a simple and welcome change. For example, you shouldn’t have to change your code because #pragma segment directives will simply be ignored if they are not appropriate. On the other hand, segmentation involves fine-tuning the memory usage of your application. You may need to rethink your memory strategy in the absence of segmentation. Also, beware of dependencies on other aspects of segmentation, such as the Segment Loader.

- Toolbox dispatching. The current toolbox dispatching mechanism (A-traps) is very 68K dependent and will certainly be different on other platforms. Trap patching is OK, but you should think carefully about your need for a patch and not try to short-circuit established patching mechanisms. Don’t assume you know the format of the trap dispatch table; use

GetTrapAddress and SetTrapAddress instead. (See also 11, “Don’t Directly Patch the ROM.”)

4. Isolate and Minimize Use of Low Memory

For as many years as the Macintosh has been shipping there have been applications that have depended on direct access to low-memory globals. Apple has said for a long time that developers must not depend on these global variables, but unfortunately there are many cases where applications *must* access these variables to function. Shared low memory is one area we must wean developers from before we can provide modern features like protected address spaces and preemptive multitasking.

Low-memory usage falls into three categories:

- Documented low-memory globals. These are the safest to use as they (or equivalent functionality) will continue to be available. For example, many applications using Standard File depend on the low-memory globals CurDirStore and SFSaveDisk. We can’t just wish those away.

- Hardware dependent low memory. Some low memory is specific to the 68K, such as exception and interrupt vectors. Dependence on these locations is bad for two reasons. First, because it will be different on future platforms, and second, because it implies supervisor level access, which may not be allowed in the future. Be very careful about depending on this kind of access. See also 8, “Don’t Depend on Interrupt Level or Supervisor Mode.”

- Undocumented low-memory globals. A lot of low memory is used by the system and has never been documented. Yet, applications persist in mucking around in them. This has always been dangerous and unsupported. Apple makes no guarantee that these globals or equivalent functionality will be available in the future.

Because many applications depend on low memory, we can’t just pull the rug from under you because every application would break. So, what’s a developer to do? *Isolate and minimize your dependence on low memory.*

If accessor functions exist, you should use them. For example, GetMBarHeight() returns the same information as the low-memory global MBarHeight.

Eventually, Apple will provide a new API that will include accessor functions for documented low-memory globals and you should use those when available. In the meantime, you might consider using macros to do the same thing. For example, to access CurDirStore you might use the C macros:

```
#define GetCurDirStore() (*(long *)CurDirStore)
#define SetCurDirStore(dirID) (*(long *)CurDirStore = (long)dirID)
```

This at least lets you isolate dependencies on CurDirStore in your source, so when Apple does change the API, you only need to change your code in one place.

5. Isolate and Minimize Use of Internal Toolbox Data Structures

If it isn’t documented, don’t use it. The data structures that are not documented aren’t documented because we expect they may change in the future. This means that if your application is dependent on any internal data structures never changing, then it is very likely that your product will break in the future. There are some major changes being considered for the Macintosh Operating System and any major change will include the internal data structures, so beware of any undocumented features on which you depend.

6. Don’t Intermix Data and Code

It will make it easier for the Operating System to move to a memory protection model if code does not write to itself. The issue is not just writing instructions into a code segment (that is, self-modifying code), but writing data into the segment. A robust operating system write-protects code segments, but Apple can’t provide that benefit to applications that depend on writing to their code segments.

The restriction does not apply to static, read-only data stored in code segments. For example, if you store strings in your code segments, but never modify them, this is not a problem.

7. Isolate Dependencies on 80 and 96-bit Extended

Different FPUs are going to have different preferred formats. On 68K, extended offers the best precision as well as the best performance and is therefore the “natural” choice. On other platforms, this may not be the case, so you should be prepared to take these differences into account. For most of you this is not a problem, but it does mean there may be differences in the size of fields in data structures and on disk, as well as in the size of parameters passed to functions. You should try to isolate and avoid dependencies on the size of floating-point numbers.

For some developers there is an issue with the perception that your applications need very high precision. There are cases where extended number precision is very useful (such as 3-D and CAD), but it is possible to make a fully 3-D graphics application that does not need extended numbers. If you find that you can not live without extended numbers, then you may find that you will need to live with slow calculations.

8. Don’t Depend on Interrupt Level or Supervisor Mode

Interrupt levels, supervisor mode, and exception handling are all very dependent on the microprocessor that is being used in the computer your code is running on. If your code thinks that it knows how to manipulate the hardware and system in supervisor mode, then it will find that it is wrong on the next generation of microprocessors. If you think your code can alter the interrupt levels, then once again you will be sadly mistaken. If your code changes the exception handlers, then your code is too dependent on the hardware on which it is running and will more than likely break on Apple’s future products.

Many applications also assume they can lock out interrupts for long periods of time. Doing so will break input/output on future systems. You should do only the most critical tasks with interrupts locked out and then only for the shortest possible time. You can also use the Deferred Task Manager to handle some interrupt level processing at a later time when there are no pending interrupts.

One common violation of supervisor mode is the use of privileged instructions. In practice, because of compatibility reasons, some instructions may be emulated, but you should not rely on that fact.

The following are the only privileged instructions that are likely to be supported in the future. Other privileged instructions will likely cause Illegal Instruction exceptions.

```
•   ORI.W       #<value>,SR     see note 1
•   ANDI.W      #<value>,SR     see note 1
•   EORI.W      #<value>,SR     see note 1
•   MOVE.W      <ea>,SR         see notes 1, 2
•   MOVE.W      SR,<ea>         see note 2
•   FSAVE       <ea>            see note 2
•   FRESTORE    <ea>            see note 2
•   RTE                         see note 1, 3
•   MOVEC.L     <Rn>,CACR       see note 4
•   MOVEC.L     CACR,<Rn>       see note 4
•   CPUSHA      BC              see note 4
```

Notes:

1. It is not possible to alter the values of either the S bit or the M bit with these instructions. The S and M bits you supply are simply ignored by the emulation of these instructions.

2. It should be possible to support all effective address modes for any instruction that uses an effective address operand providing that mode is legal for this instruction. Use of illegal addressing modes results in an Illegal Instruction exception.

3. Only normal four-word frames are supported by the emulation of RTE. Other frame kinds generate Illegal Instruction exceptions.

4. Use of `MOVEC` (or `CPUSH` and `CINV` on 68040 machines) to control the instruction and data caches is strongly discouraged. Developers should use system software routines such as `FlushInstructionCache()` to accomplish the same thing.

9. 32-bit Clean Mandatory

Apple has been saying for many years that all applications need to be able to run in a 32-bit address environment. Not only is this important because it allows users to take full advantage of the memory of the machine, but it allows Apple to transition from one memory model (24-bit) to another memory model (32-bit). Supporting two different memory models has allowed Apple to maintain compatibility, but the cost has been a lot of extra code in the ROM and some performance penalty. So now we’re preparing to take the next step—32-bit only addressing.

Most developers have by now removed any dependency on a 24-bit environment, but some developers have cheated a bit. Rather than making use of the upper 8-bits of an address they use the upper bit, on the assumption (valid for the most part) that everything happens in the lower 2 gigabytes of the 4 gigabyte address space. In other words, they are only 31-bit clean. This is very bad as future operating system releases will take full advantage of the 4 gigabyte range of addresses.

The message is, not only will we be 32-bit only, but we really mean 32-bit.

10. Don’t Hit the Hardware Directly

Apple has warned developers for a long time to lay off the hardware. You should always access hardware indirectly, through a manager or driver. For example, some applications persist in generating sound by accessing the old sound buffer pointed to by the SoundBase low-memory global. Although this technique has been supported in the past, its use has not been recommended since the introduction of the Sound Manager with the Macintosh II. Very soon it will not work at all. Use the Sound Manager instead.

We realize developers sometimes feel they need to access the hardware directly to squeeze the most performance out of the machine. But doing so severely limits Apple’s ability to advance the hardware platform and it hurts customers in the long run because your applications break when new hardware is released.

Think twice before accessing the hardware directly. The pain will probably outweigh the gain.

11. Don’t Directly Patch the ROM

Do not access the trap tables directly. Use `SetTrapAddress` and `GetTrapAddress` to guarantee future compatibility. The next major ROM will use a vectorization method for patching. Vectorization is definitely going to affect the trap tables in the future. In the near future, we may even bypass the trap table and make `Set/GetTrapAddress` use the vector locations.

Below is a sample of the correct way to patch a trap. This code is from the 7.0 sample for making an INIT/cdev.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;MACRO
;      ChangeTrap    &trap, &type, &newAddress, &oldAddressStore
;
;      Macro used to change the trap address and optionally save the old
;      routine's address. You pass in the trap number of the trap you want
;      to patch, the type of the trap (newTool or newOS), the address of the
;      routine to store in the trap table, and a pointer to a memory location
;      that will hold the old address. If &oldAddressStore is some value other
;      than NIL, this macro will get the old trap's address and save it there.
;
;      NOTE:  This macro translates &newAddress and &oldAddressStore into
;             their new locations. To do this, it relies on A1 pointing to
;             the block in the system heap that holds the patch, and for
;             FirstResByte to be defined.
;
;Input:
;      A1: address of patch code in system heap
```

```
;
;Output:
;       oldAddressStore: address of old trap routine
;       D0, A0 are destroyed.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        MACRO
        ChangeTrap     &trap,&type,&newAddress,&oldAddressStore

        IF (&oldAddressStore ≠ 'NIL') THEN

                move.w #&trap,D0
                _GetTrapAddress ,&type
                move.l A0,&oldAddressStore-FirstResByte(A1)

        ENDIF

                move.w #&trap,D0
                lea           &newAddress-FirstResByte(A1),A0
                _SetTrapAddress ,&type

        ENDM
```

An alternate piece of sample code that works very well for Think C follows:

```
#include <SetUpA4.h>


/*
 *generic code patch loader
 *MacDTS/pvh
 *©1989-90 Apple Computer, Inc.
 *Copies CODE resource into system heap.  puts original trap address
 *at head of code in case you need it later.  This is THINK C specific of course.
*/

#define GetNextEvent_trap    0xA970

/*
 *this is needed because the QuickDraw global thePort is not defined at INIT time
*/
typedef struct myQDGlobalsDef {
        char           qdPrivates[76];
        long           randSeed;
        BitMap         screenBits;
        Cursor         arrow;
        Pattern        dkGray;
        Pattern        ltGray;
        Pattern        gray;
        Pattern        black;
        Pattern        white;
        GrafPtr        thePort;
        } myQDGlobalsDef;

/* OS traps start with A0, Tool with A8 or AA. */
short GetTrapType(short theTrap)
{
        if((theTrap & 0x0800) == 0)         /* per D.A */
                return (OSTrap);
        else
                return (ToolTrap);
```

```
}

/*
 *The INIT
*/
void main()
{
       Handle              h;
       long                size;
       Ptr                 codeSpot;
       KeyMap              theKeyMap;
       myQDGlobalsDef      myQDGlobals;
       long                trapAddr;
       long                *blah;

       RememberA0();
       SetUpA4();

       InitGraf(&myQDGlobals.thePort);
       GetKeys(&theKeyMap);

       /* don't install if shift key is down */
       if(theKeyMap.Key[1] != 1L)   {
              h = GetResource('CODE', 12);        /* our patch code resource */

              if (h != 0L) {
                     size = SizeResource(h);

                     /* size of our resource + 4 for saved trap address */
                     codeSpot = NewPtrSys(size+4L);
                     HLock(h);      /* lock the resource handle just because */
                     /* move in the CODE resource into the SYS heap */
                     BlockMove(*h, codeSpot+4L, size);
                     HUnlock(h);    /* unlock it */

                     /* get the 'current' trap address */
                     trapAddr = NGetTrapAddress(GetNextEvent_trap,
                            GetTrapType(GetNextEvent_trap));

                     /* this is skanky but move the original trap */
                     blah = (long *) codeSpot;
                     /*address into the top of the new block.  Would */
                     *blah = (long) trapAddr;
                     /* set to the new trap address */
                     NSetTrapAddress(codeSpot + 4L, GetNextEvent_trap,
                            GetTrapType(GetNextEvent_trap));



/*
; …or use this assembler source if you'd rather
asm {
       move.l size, d0      ; size of our resource
       add.l  #4, d0        ; add 4 to size for place holder for real trap address
       _NewPtr       SYS    ; create block in system heap
       move.l a0, codeSpot  ; spot to save pointer

       move.l h, a0
       _HLock               ; lock the handle of our code

       move.l size, d0      ; size of our resource
       move.l codeSpot, a1  ; head of block
```

```
        adda.l #4, a1         ; we want the first 4 bytes for saving original trap
                              ; address
        move.l h, a0          ; handle to our patch code
        move.l (a0), a0       ; dereference to actual address
        _BlockMove            ; move it in

        move.l h, a0
        _HUnlock              ; unlock the handle

; save the real trap address in 4 byte spot at head of block
        move.w #GetNextEvent_trap, d0
        _GetTrapAddress
        move.l codeSpot, a1
        move.l a0, (a1)
; set the trap address to our patch, 4 bytes past the block header
; (remember the first 4 bytes our the saved original address)
        move.w #GetNextEvent_trap, d0
        move.l codeSpot, a0
        adda.l #4, a0
        _SetTrapAddress
}
*/
                      }
               }
        RestoreA4();
}
```

Sample patch code for _GetNextEvent using THINK C (there is a little assembly involved):
This would be compiled in a separate project as CODE resource ID=12

```
pascal Boolean main(short mask, EventRecord *evt)
{
        long   realTrapAddr ;

        asm {
               movem.l d3-d7/a2-a6,-(sp)
               ; THINK C kindly places pointer to top of CODE resource in a0
               ; we saved the original trap address just ahead of our patch so
               ; let's go get it and save it
               move.l a0, a1
               sub.l  #4, a1
               move.l (a1), realTrapAddr
               }

        /* do the THINK stuff *
        RememberA0();
        SetUpA4();

        SysBeep(1);

        /* do the THINK stuff *
        RestoreA4();

        asm {
               movem.l (sp)+, d3-d7/a2-a6
               move.l realTrapAddr, a0     ; get original trap address
               unlk   a6                                 ; unlink
               jmp    (a0)                               ; and jump to it
               }
}
```

12. Don’t Depend on Resources Being in the System File

In a future version of the ROM we plan to put parts of System 7 into the ROM. This means that if you expect to see a resource in the System file, then you may find yourself in trouble. Some System 7 resources and packages will soon reside in the ROM.

To specifically request resources from the ROM, you must set the low-memory global RomMapInsert (*Inside Macintosh* Volume VI, page VI-19) as follows:

```
// Accessor functions to isolate use of low memory
//
#define GetROMMapInsert() (*(short *)RomMapInsert)
#define SetROMMapInsert(mapValue) (*(short *)RomMapInsert = (long)mapValue)

SetROMMapInsert(mapTrue);
resHdle = GetResource (resType, resID); /* Get the resource (clears RomMapInsert) */
```

If you want to search both the system file and the ROM for a resource, you can use `RGetResource` (*Inside Macintosh* Volume V, page V-30).

13. Don’t Make Assumptions About the Contents and Size of ROM

The current Quadra ROMs are 1 megabyte in size. As we add portions of System 7 to ROM, they will grow even bigger. Not only will they grow, but things will be rearranged in the ROM. Don’t assume you know the layout or location of entry points in the ROM. If you’re dependent on knowing the ROM version number you may break on future ROMs.

14. SCSI

There are some things that you should already know about SCSI, but it is important to reiterate some potential problems that you may have already experienced and probably will continue to see in the future.

SCSIStat:

Although the `SCSIStat` routine has been provided in the SCSI Manager ever since the Macintosh Plus, it is a mostly useless routine. Although it does provide some information on all the Macintosh computers until the Quadra models, this information has encouraged some developers to design products that did not follow the SCSI specification. If your device is a true SCSI device, then the device and its driver should never care what state the SCSI bus is in. Some developers found on the Quadra computers that we changed the hardware that we were using to improve the performance and lead Apple’s hardware into the future of SCSI. This change caused several developers to have problems, in part because the new hardware provided no way for the call `SCSIStat` to perform with anything that resembled accuracy or reliability.

Protocol:

Another problem that some developers had with SCSI on the Quadra computers was related to their not following the SCSI specification on protocol. The Target is in control of the bus and all the driver does is call the appropriate routines in the correct order. If you depend on these routines being synchronous and providing feedback as they are being called, then your product had problems on the Quadra computers. On the Quadra models, the SCSI operations are queued up until the `SCSICmd` is sent. At this time the SCSI transaction is done in one big operation. There are no errors reported along the way, so the drivers must operate on the assumption that the SCSI Manager knows what it is doing and that the hardware knows what it is doing.

It is very important to follow the SCSI specification.

Patches:

If your products patch any part of the SCSI Manager at this time, be aware that the SCSI Manager is soon going to go through a major overhaul. Your patches may cause problems with the new SCSI Manager, so keep an eye out for all documentation that Apple generates on the SCSI Manager. If you feel that you may need some advance warning about the new SCSI Manager, then we highly recommend that you contact Apple Evangelism and make sure they understand that you feel you might have problems with the new SCSI Manager and that you would like to receive any documentation they have when it is available.

15. VIAs

There are some third-party products that like to interrogate the VIAs for information about ADB activity or get high-resolution timing from the VIAs. There are other products that use the VIAs to change the interrupt levels of the hardware. Any products that depend on the VIAs in any way are going to have a great deal of trouble in the future as Apple hardware gradually depends less and less on having VIAs. You may have started to see some of the Macintosh computers either eliminating one of the VIAs or emulating the VIAs in a big ASIC. As the hardware continues to evolve, these VIAs will become even less available.

16. Do the Right Thing

“WHY ASK WHY,” “JUST DO IT”

If you need to find out if something is available for your use, then use Gestalt to find out. If Gestalt does not tell you what you need to know, then you may be looking for something that you should not be worrying about. There are some cases where this is not true, but in most cases it is.

It is also important that you look for the exact information that you need to know. If you want to know if there is an FPU, then check for that—nothing else will tell you the truth.

Last, but not least—**when in doubt *ask*!**

Further Reference

- Technical Note DV 24 - Fear No SCSI