About Compatibility

Compatibility is the ability of a program to execute properly in different operating environments. Compatibility is important if you want to write software that runs, with little or no modification, on all members of the Macintosh family and in all system software versions. If you want to take advantage of particular software or hardware features that may not be present on all Macintosh computers, you need to know how to determine when those features are available.

To appreciate why compatibility is a real concern, imagine that from all the Macintosh computers currently in operation in the world, you were to choose two at random. You would quite likely find a number of differences in the hardware and software configurations of those two machines. You might find different CPUs, different memory management units (MMUs), different amounts of RAM, different shapes and sizes of monitors, and so forth. You are also likely to find different versions of system software, different ROM versions, different AppleTalk drivers, different versions of managers, different printer interfaces, etc. Ideally, you want your product to run on both of those machines, regardless of the many significant differences between them. If you succeed in writing your application so that it does operate on both of those machines, you have succeeded in writing compatible software.

Fortunately, it is possible to write software that is compatible across the entire Macintosh line of computers. Apple's **Compatibility Guidelines** provide a number of guidelines that you should follow if you want your applications to run on the greatest number of Macintosh computers. Some of these guidelines are quite general and apply to all programs; some apply only if you are programming in assembly language.

One key to achieving compatibility is not to depend on things that may change. As the **Operating System** and **User Interface Toolbox** evolve to accommodate the needs of developers and users, many of their elements will vary. Whenever possible, Apple strives to add features without altering existing interfaces. In general, you can assume that Operating System and Toolbox routines are less likely to change than data structures. Therefore, you should never directly manipulate data structures that are internal to a manager or system software routine, even if their structure is documented. Instead, you should manipulate those structures only indirectly, by calling **Operating System** and **User Interface Toolbox** routines that achieve the desired effect. In particular, you should never alter any portion of a data structure marked as unused or reserved.

Another key to writing compatible code is to code defensively. Do not assume that users perform actions in a particular order, and do not assume that function and procedure calls always succeed. You should always test the return values of routines for errors, as illustrated in most of the code samples.

Using Memory Wisely

A major cause of compatibility problems, especially in connection with applications running in the A/UX operating system, is misuse of the **Memory Manager**. Here are some important points to keep in mind:

- Do not set or clear bits in master pointers directly. Use the **Memory Manager** traps (for example, **HLock**) instead.

- Always check the handle or pointer returned by a routine to make certain that it is not NIL. A NIL handle may indicate that a memory allocation failed or that a requested resource could not be found.

- Always check that a handle marked as purgeable has not been purged before using that handle. You can check for a purged handle like this:

```
if (*myHandle ! = NIL)      handle not purged
```

- Do not create your own handles; instead, use the **Memory Manager** function **NewHandle**.

- Never make assumptions about the contents of **Memory Manager** data structures.

If you have followed all these guidelines, it is likely that your application is 32-bit clean; that is, it operates correctly in an environment where all 32 bits of handles and pointers are used to store memory addresses. When running with 32-bit addressing in system 7.0 and A/UX, your applications must be 32-bit clean or they may not operate correctly. See the **Memory Manager** for more information about these issues.

Using Assembly Language

In general, your software should not include 68000 instructions that require the processor to be in supervisor mode; these include instructions that modify the contents of the Status Register (SR). Do not modify the SR as a means of changing the Condition Code Register (CCR) half of the SR; instead, use an instruction that addresses the CCR directly. Do not use the User Stack Pointer or turn interrupts on and off.

If you wish to handle your own exceptions (thereby relying on the position of data in the exception's local stack frame), be aware that exception stack frames vary within the 68000 family.

In particular, don't use the TRAP instruction. Also, the Macintosh SE and Macintosh II hardware does not support the TAS instruction, which uses a special read-modify-write memory cycle.

Some Macintosh computers use memory protection and may prevent code from writing to addresses within code segments. Also, the 68020 and 68030 cache code as it is encountered. You should allocate data blocks on the stack or in heap blocks separate from the code, and your code should not modify itself.

Accessing Hardware

You should never address hardware directly; whenever possible, use the routines provided by the various device drivers and managers to send data to the available hardware. The addresses of memory-mapped hardware (like the VIA1, VIA2, SCC, and so forth) are always subject to change, as is the hardware itself. More important, direct access to such hardware is not possible in every operating environment. In multi-user systems like A/UX, for instance, the operating system manipulates all hardware; applications simply cannot write directly to hardware addresses.

You should also avoid writing directly to the screen. Use **QuickDraw** routines whenever possible to draw on the screen. If you absolutely must write directly to the screen, do not assume that the screen is a fixed size or that it is in a fixed location. The location, size, and bit depth of the screen differ in various machines. On machines without **Color QuickDraw**, you can use the **QuickDraw** global variables screenBits.bounds to determine the size of the main screen, screenBits.baseAddr to determine the start of the main screen, and screenBits.rowBytes to determine the offset between rows. On machines with **Color QuickDraw**, the device list (described in the Graphics devices) tells the location, size, and bit depth of each screen; screenBits contains the location and size of the main device; and the global variable GrayRgn contains a region describing the shape and size of the desktop.

Using Low-Memory Global Variables

Don't rely on low-memory global variables. Many of these variables have been previously documented in *Inside Macintosh,* but many have not. In particular, you must avoid undocumented low-memory global variables because they are most likely to change. But you should try to avoid even well-known global variables because they may not be available in all environments or in the future. In general, you can avoid using low-memory global variables by using available routines that return the same information. (For example, the **TickCount** function returns the same value that is contained in the low-memory global variable Ticks.)

Determining Whether a Trap Is Available

One important way that the Operating System and Toolbox have changed through successive versions of the ROM and system software is by the addition of numerous new traps. For example, the **Time Manager** released with system 7.0 includes a new trap, **InsXTime**, that provides certain improvements over the existing trap, **InsTime**. By using **InsXTime** instead of **InsTime**, your application can ensure that the periodic actions it requests execute at a fixed frequency that does not drift over time. Before using a trap that is not available on all machines, however, you need to determine whether it is available; if you call **InsXTime** on a machine that does not implement it, your program will crash.

There are several ways your application can check the availability of a particular trap. First, you can call the **Gestalt** function that is discussed to see if the appropriate version of the corresponding driver or manager is available. For example, the trap **InsXTime** is included in the extended **Time Manager** but not in earlier versions of the **Time Manager**. So you could use **Gestalt** to determine which version of the **Time Manager** is available in the current operating environment. If **Gestalt** reports that the extended **Time Manager** is present, you can safely call **InsXTime** to queue your request.

There are several cases, however, in which you cannot use **Gestalt** to determine whether a specific trap is implemented. You cannot, for instance, use **Gestalt** to determine whether the **Gestalt** trap itself is available. In addition, the trap whose existence you wish to test might not be included in any manager or, if it is, there might not be a **Gestalt** selector code for that manager. The **WaitNextEvent** trap is a good example of this: there is no way, using **Gestalt**, to determine whether **WaitNextEvent** is available.

A second way to determine the availability of a particular Operating System or Toolbox trap is by testing directly for the existence of the trap, using the technique illustrated in the listing below. You should use this method to test whether **Gestalt** is available before calling **Gestalt**. You should also use it to test for the existence of traps not included in managers or drivers about which **Gestalt** can report. This listing illustrates how to test the availability of **WaitNextEvent**.

```
#include<Traps.h>
#include<OSUtils.h>

#define TrapMask 0x0800

short NumToolboxTraps( void )
{
    if (NGetTrapAddress(_InitGraf, ToolTrap) ==
            NGetTrapAddress(0xAA6E, ToolTrap))
        return(0x0200);
    else
        return(0x0400);
}

TrapType GetTrapType(short theTrap)
{

    if ((theTrap & TrapMask) > 0)
        return(ToolTrap);
    else
        return(OSTrap);

}

Boolean TrapAvailable(short theTrap)
{

    TrapType  tType;

    tType = GetTrapType(theTrap);
    if (tType == ToolTrap)
    theTrap = theTrap & 0x07FF;
    if (theTrap >= NumToolboxTraps())
        theTrap = _Unimplemented;

    return (NGetTrapAddress(theTrap, tType) !=
            NGetTrapAddress(_Unimplemented, ToolTrap));
}

Boolean WNEAvailable(void)
{
    return TrapAvailable(_WaitNextEvent);
}
```

The **NumToolboxTraps** function relies on the fact that the **InitGraf** trap (trap number 0xA86E) is always implemented. If the trap dispatch table is large enough (that is, has more than 0x200 entries), then 0xAA6E always points to either *Unimplemented* or something else, but never to **InitGraf**. As a result, you can check the size of the trap dispatch table by checking to see if the address of trap 0xA86E is the same as 0xAA6E. After receiving the information about the size of the dispatch table, the **TrapAvailable** function first checks to see if the trap to be tested has a trap number greater than the total number of traps available on the machine. If so, it sets the *theTrap* variable to *Unimplemented* before testing it against the *Unimplemented* trap.

Note: The technique presented in the listing above for determining whether a particular trap is available differs from techniques formerly supported by Apple. The previous method determined the size of the trap dispatch table by checking the machine type. This type of check should not be used for any purposes other than simply displaying the information, as explained in **Using the Gestalt Manager** .