Macintosh® Programmer's Guide to MultiFinder™

APDA™

June 3, 1988

Contents

Preface

Welcome to MultiFinder

Programmer's Guide to MultiFinder introduces MultiFinder™, a new set of operating system functions designed to increase the efficiency and functionality of the Macintosh® family of computers. In addition, it outlines specific programming guidelines intended to help software developers write MultiFinder-compatible applications.

This book is directed toward the proficient Macintosh application developer. For an introductory discussion on how to write a Macintosh application, consult the introductory volumes of the *Macintosh Technical Library*.

This is not a dedicated reference manual. While the Macintosh programming paradigm remains largely unchanged, MultiFinder does reflect a departure from the original one-application-open-at-a-time desktop environment. To appreciate the nuances of how MultiFinder works and to achieve the highest degree of compatibility with MultiFinder—present and future versions—you should read this entire book. Take a moment to scan the section "Some Conventions Used in This Manual"; this section is particularly important in this book. *Programmer's Guide to MultiFinder* includes

- a description of the traditional Macintosh user's model, the MultiFinder user's model, and the MultiFinder programmer's model
- definitions of the three types of MultiFinder-friendly applications: well-behaved, MultiFinder-aware, and special-purpose
- detailed descriptions of the new Macintosh programming features: the SIZE resource, the event call WaitNextEvent, the Notification Manager, suspend and resume events, and the new temporary memory allocation calls
- descriptions of cooperative multitasking and background notification
- programming guidelines for ensuring MultiFinder compatibility
- guidelines for designing well-behaved, MultiFinder-aware, and special-purpose applications

- code examples for handling the new suspend and resume events, saving A5 and CurrentA5, writing completion routines for asynchronous Device Manager calls, determining whether WaitNextEvent and the temporary memory allocation calls are implemented, and writing MultiFinder-aware applications

Programmer's Guide to MultiFinder does not include information about

- programming in general
- getting started as a developer

To use this book, you should already be familiar with the information that's in *Inside Macintosh* and have experience writing Macintosh applications.

For information about becoming a developer or obtaining developer support, write to

Developer Programs
Mail Stop 51-T
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014

Organization of this manual

This manual is organized as follows:

- Chapter 1, "Introduction to MultiFinder," describes the traditional Macintosh user's model, the MultiFinder user's model, the MultiFinder programmer's model, and the different types of MultiFinder-friendly applications.
- Chapter 2, "Well-Behaved Applications," describes the characteristics of applications that will work well in the MultiFinder environment. Included here are general programming guidelines for the user interface and memory management, as well as a detailed discussion of the A5 world and some miscellaneous programming hints.
- Chapter 3, "MultiFinder-Aware Applications," introduces the concept of an application that takes maximum advantage of the background processing time now available under MultiFinder. Also included in this chapter are descriptions of the SIZE resource; the two new types of app4Evts, suspend and resume; WaitNextEvent; the new temporary memory allocation calls; and launching and sublaunching.
- Chapter 4, "Special-Purpose Applications," describes three different special applications: embedded services, faceless background tasks, and desk accessories.
- Appendix A, "A C Example of a MultiFinder-Aware Application," gives a C example of an application that is MultiFinder-aware.
- Appendix B, "A Pascal Example of a MultiFinder-Aware Application," gives a Pascal example of an application that is MultiFinder-aware.

- Appendix C, "Resource Descriptions for the Example MultiFinder-Aware Application," lists the necessary resource descriptions for the MPW™ Rez tool used in the C and Pascal program examples of a MultiFinder-aware application.
- Appendix D, "The Notification Manager," provides a detailed description of this new manager.
- Appendix E, "A Summary of the MultiFinder Traps," summarizes the new traps available to applications under MultiFinder.

About the Apple Programmer's and Developer's Association

The Apple Programmer's and Developer's Association (APDA™) is an excellent source of technical information for anyone interested in developing Apple-compatible products. Membership in the association allows you to purchase Apple technical documentation, programming tools, and utilities. For information on membership fees, available products, and prices, please contact

APDA
290 SW 43rd Street
Renton, WA 98055
(206) 251-6548
AppleLink: APDA
MCI: 312-7449
CompuServe# 73527,27

Some conventions used in this manual

The following conventions have been adopted for use in this book:

- The **A5 world** consists of an application's own global variables and its private set of QuickDraw globals (both are accessed through A5), the set of low-memory globals associated with the application by MultiFinder, and the application's heap and stack. The single-Finder™ environment only allowed one A5 world at any given time. With MultiFinder active, there is an A5 world for each open application.
- **Cooperative multitasking** is the result of a foreground application and one or more applications concurrently running in the background interactively and sharing a limited amount of resources.
- An **embedded service** is a special-purpose application that runs only in the background.

- **Event calls** refer to GetNextEvent, WaitNextEvent (see Chapter 3 for a detailed description of this new event call), and EventAvail.
- A **faceless background task** is another special-purpose application that is almost invisible. It is minimal in size and has no user interface—no icon will appear in the Apple menu and no windows will be displayed.
- A **MultiFinder-aware application** is one that calls WaitNextEvent, handles suspend/resume events, specifies a SIZE resource, and optionally performs some background work without significantly affecting the responsive nature of any application running in the foreground.
- **Null event processing time** is the time when most applications sit idle because there are no events initiated by the user or no windows to be redrawn.
- A **well-behaved application** is one that follows the programming guidelines outlined in Chapter 2.

Chapter 1

Introduction to MultiFinder

This chapter introduces the new set of multitasking operating system functions for Macintosh® computers, MultiFinder™. The name "MultiFinder" is actually misleading—MultiFinder is *not* part of the Finder™. Instead, it is a set of additional operating system functions designed to allow an increased level of functionality with a minimal impact on the Macintosh programming model. MultiFinder is compatible with the Macintosh Plus, Macintosh SE, and Macintosh II computers and resides in the System Folder.

The additional functionality MultiFinder provides is essentially the ability to run multiple applications on the Macintosh, all of them sharing the available desktop. Great care has been taken to introduce this new functionality without noticeably altering the familiar Macintosh desktop.

You should take note of a couple of important issues here. First, at present, a user retains the ability to turn MultiFinder off or not run it at all—reserving single-application capability if desired. Second, MultiFinder will eventually represent the exclusive Macintosh desktop environment.

This chapter provides an overview of the traditional Macintosh desktop and the new MultiFinder desktop environments, and explains cooperative multitasking and its role in the new MultiFinder programming model. Finally, this chapter ends with a discussion of the different types of Macintosh applications.

The traditional Macintosh user's model

The original Macintosh user interface presented a powerful metaphor of the everyday desktop to both the Macintosh customer and software developer. The Macintosh user has an enormous amount of freedom to customize a personal desktop work space and clearly benefits from the high degree of compatibility among Macintosh applications.

Each application running in the Macintosh single-Finder environment has control of the entire desktop because it knows the state of the entire Macintosh machine. As part of this original implementation, the desktop disappears while an application runs, and only one application can run during each work session.

The MultiFinder user's model

MultiFinder introduces some welcome changes to the Macintosh desktop environment. Users can have any number of applications open at the same time, including the Finder, and can easily switch between them.

For example, a user could have a word processing program, an accounting package, a communications program, and HyperCard® all open simultaneously.

When running under MultiFinder, the Finder is not closed when another application is opened. The user can activate the Finder or any open application in one of three ways: by clicking the appropriate name and icon in the Apple menu, by clicking the small icon in the upper-right corner of the menu bar (until the correct icon appears), or by clicking the desired window on the desktop.

The desktop metaphor remains

MultiFinder has not restricted or limited the familiar desktop metaphor. Since many applications can now be open at the same time, their windows can overlap each other and users can quickly switch between them—MultiFinder actually models a real working desktop better than the single-Finder environment does.

Applications, windows, and the menu bar

When a user opens an application under MultiFinder, that application becomes active and begins running in the foreground. The menu bar will contain the active application's menu titles and a small icon that represents the application in the rightmost corner of the menu bar. Also, all the active application's windows will come to the frontmost layer of the desktop.

For example, a user could have a word-processing application running in the foreground with three windows open for three different documents, as well as a number of other applications open simultaneously. If the user brings one of the other applications to the foreground, performs some work, and then once again brings the word-processing application to the foreground, all three of its windows will come to the frontmost layer of the desktop.

The MultiFinder programmer's model

The user interface is the most important element of the Macintosh environment. Although the MultiFinder engineers have taken great care to preserve the look and feel of the Macintosh interface, a similar responsibility also rests with developers who will write-MultiFinder-compatible applications. While MultiFinder has dramatically extended the user interface of the Macintosh, the programming model remains basically intact. MultiFinder supports a new concept for the Macintosh—**background processing**—that allows applications not currently running in the foreground to make use of processing time that is unused by the current foreground application.

MultiFinder uses this **null event processing time**—the time when most applications sit idle because there are no events initiated by the user or no windows to be redrawn—to allow other applications not running in the foreground to perform useful work. The Finder, for example, now uses this time to keep the view within its windows and on the desktop consistent (disk insertions, files being added or removed, and so on). Since this time is essentially wasted in the single-Finder environment, MultiFinder does not inhibit or noticeably slow the response time of the application running in the foreground.

In addition, if an application wants to take advantage of this unused null event processing time and perform some work in the background, it must take on some additional responsibility. Applications running in the background must know how to coexist with foreground applications because the user is interacting with the foreground application and expects it to respond immediately.

Cooperative multitasking

Cooperative multitasking is the result of a foreground application and one or more applications concurrently running in the background interactively sharing a limited amount of resources—the sort of kindness you might expect among individuals with good manners.

While most operating systems regulate this "sharing" by having the *system* parcel out control, MultiFinder relies on the willingness of foreground and background *applications* to share the available resources. (MultiFinder does regulate the sharing of most resources, including microprocessor time; however, the application is allowed to decide when it will give up control of the microprocessor.) This "kindness of strangers" philosophy between applications running in the foreground and those running in the background forms the basis for MultiFinder's friendly cooperation.

The burden of responsibility for this sharing of resources lies with all applications. Due to the cooperative nature of MultiFinder, if one application holds on to the microprocessor too long, the other applications will appear unresponsive.

Background processing and event calls

Applications running in the background receive processing time when the foreground application releases the microprocessor with a call to an event call and no events for the foreground application are pending. Exceptions to this rule are mouse movements outside a predefined area, and after a predefined sleep value set by the foreground application has expired (see the description of WaitNextEvent in Chapter 3, "MultiFinder-Aware Applications," for more information on how your application can take advantage of these new programmer-defined parameters).

Background notification

Applications running in the background cannot use the standard methods of communicating with the user, such as alert or dialog boxes, since such windows wouldn't necessarily be visible under the windows of the foreground application. Also, in the single-Finder environment, some applications used nonstandard and unsupported notification techniques. Now, under MultiFinder, if something occurs that requires the user's immediate attention, applications should use the **Notification Manager** (see Appendix D) to notify the user.

It is suggested that your application adopt a three-level **notification hierarchy** for communicating with the user as a user interface standard:

1. A diamond displayed next to the application's name in the Apple menu.
2. An icon for the application alternating with the Apple icon menu title in the menu bar (and any other background-resident application icons that need attention), and a diamond displayed next to the application's name in the Apple menu.
3. Both the application's icon and the diamond are displayed, and an alert box is displayed on the frontmost layer to notify the user that something needs to be done.

The user should be allowed to set the desired level of notification in a dialog window (for example, PrintMonitor's Preferences dialog window). The default should be level two (display the background application's icon in the menu bar and display a diamond next to the application's name in the Apple menu). Sound can also be used for levels two and three; however, the user should have the option of turning it off. Sound, alert boxes, and icon use should all be optional. The most important idea here is that the user should have the final say in how the notification process will work.

Users should also be able to turn background notification off altogether, except in cases where damage would occur or data would be lost (for example, a file server going down in two minutes). Background-resident applications should not do anything that might affect the foreground application, such as changing the pointer or altering the menu bar.

The three types of Macintosh applications

Applications generally fit into three categories while running under MultiFinder. Well-behaved applications work fine—they just don't take advantage of the new expanded functionality. MultiFinder-aware applications take advantage of some or all of MultiFinder's new specific capabilities—perhaps doing some work in the background. Finally, special-purpose applications perform most or all of their work in the background. While desk accessories are not applications, they represent a special case and will be discussed at the end of Chapter 4, "Special-Purpose Applications."

Well-behaved applications

A **well-behaved application** is one that generally follows the standard Macintosh programming procedures outlined in *Inside Macintosh*.

Applications that write directly to the screen, directly modify Window Manager or Menu Manager data structures, rely on the contents of low memory, or use other shortcuts to save time are not compatible with MultiFinder.

A well-behaved application regularly makes an **event call** (GetNextEvent, WaitNextEvent, or EventAvail) to provide frequent times when the application may be suspended, follows the standard Macintosh notification procedures, and can function properly anywhere in memory.

See Chapter 2 for further information on well-behaved applications.

MultiFinder-aware applications

A **MultiFinder-aware application** is one that handles suspend and resume events, calls WaitNextEvent, includes a SIZE resource (see Chapter 3 for a detailed description of these three new programming features), and uses normally unused null events for effective background work, while not significantly affecting the responsive nature of the application running in the foreground.

Faster switching

A MultiFinder-aware application can speed up the switching process by being responsible for converting its own private Clipboard when a user clicks on another application. By following the new programming guidelines outlined in Chapter 3, "MultiFinder-Aware Applications," applications can ensure that switching between open applications will be even faster.

A better citizen

To be considered truly MultiFinder-aware, applications should call WaitNextEvent to allow other applications to use any unused processing time. Since MultiFinder allows many applications to share the available resources, if you don't need them, allow someone else the opportunity. Because certain MultiFinder-unfriendly applications may not call WaitNextEvent, applications running in the background cannot be guaranteed any microprocessor time.

Supporting the responsive nature of the foreground application is an important issue for applications running in the background. The foreground application, however, is not required to provide the same service to other applications. The idea here is not to slow down the responsive nature of the foreground application, but rather to allow other applications the opportunity to make use of time that would normally be wasted.

More flexible memory management

Applications should not depend on running in a particular area of memory and should not require large amounts of memory to function properly just because they were given control of the entire machine in the single-Finder environment.

MultiFinder does provide a means for applications to get additional memory; however, this memory should only be used for very short-term needs and should be returned as soon as possible. It is not to be used for long-term storage (see Chapter 3, "MultiFinder-Aware Applications," for more information on these temporary memory allocation calls).

Special-purpose applications

Embedded services, faceless background tasks, and desk accessories (for the purpose of this manual) each represent different types of special applications that will run with MultiFinder. See Chapter 4, "Special-Purpose Applications," for more information on these applications.

Embedded services

An **embedded service** runs only in the background. This type of application is normally not visible and interacts heavily with the Notification Manager (see Appendix D).

One example of an embedded service is PrintMonitor—a background printing utility supplied with MultiFinder.

Faceless background tasks

A **faceless background task** is almost invisible. It is minimal in size and has no user interface—no icon will appear in the Apple menu, no windows will be displayed, and no port exists. If any user interaction is required, it uses the Notification Manager.

A faceless background task sets the **canBackground** and **backgroundOnly** bits in the **SIZE resource** (see Chapter 3 for more information on the SIZE resource) and can't display a user interface.

A good example of a faceless background task that looks for printer spool files is BackGrounder.

Desk accessories

MultiFinder has eliminated the unique advantages that gave desk accessories increased functionality over applications. While it's true that MultiFinder continues to support the standard desk accessory model, you are encouraged to write small applications in the future.

It's important to note here that desk accessories are now loaded into the system heap instead of the application heap (except when the Option key is held down). Therefore, desk accessories that rely on being loaded in a specific application's heap will not function properly under MultiFinder.

Chapter 2

Well-Behaved Applications

If you have been following the programming guidelines specified in *Inside Macintosh*, your application will probably work as expected under MultiFinder. This chapter outlines a number of programming guidelines that applications should follow to ensure future compatibility with MultiFinder.

A **well-behaved application** regularly makes an **event call** (GetNextEvent, WaitNextEvent, or EventAvail) allowing for frequent suspension times, follows the standard Macintosh notification procedures, can function properly anywhere in memory, and follows the other guidelines specified in this chapter.

Be aware that MultiFinder allows special types of applications: applications that perform part of their work in the background while another application is running in the foreground, and applications that do all their work in the background. A well-behaved application should make event calls to ensure that such applications will be able to use any null event processing time.

Windows, menus, and screens

Save your window positions; there is nothing more irritating to the Macintosh user who sets up a MultiFinder work space on the desktop than to have to reposition the windows of applications every time the Macintosh is turned on.

Applications that lay out their control panels and palettes in separate windows need to be careful of "gaps" in the layout. If a user accidentally clicks in one of these gaps, another application could be switched to the foreground unintentionally.

The user should maintain control over the initial positioning of free-floating palettes (for example, the MacPaint® 2.0 Command-T option that allows the user to position the Tool palette right at the present location of the cursor). Otherwise, the initial position of these palettes should be within a few pixels of the window. Mini-windows and tear-off menus should disappear when an application is switched out and reappear when the application returns to the foreground.

❖ *Note:* MultiFinder suspends a well-behaved application (which isn't aware of MultiFinder) by creating a situation similiar to that which occurs when a user opens a desk accessory; that is, the application receives a deactivate event for its front window. Similarly, MultiFinder causes a well-behaved application to resume by sending the application an activate event.

There have always been a number of suggested "don'ts" connected with data structure manipulation—now, under MultiFinder, these suggestions are no longer optional.

Don't modify Window Manager data structures directly

Don't let your application modify Window Manager data structures directly. The Window Manager owns the Window Manager data structures. These include all the low-memory values defined by the Window Manager, in addition to any of the fields (including the grafPort fields) contained in the window record itself. Because the procedural interface to the Window Manager is so effective, direct data structure modification is rarely done (one exception is windowKind), but beware nonetheless.

Because MultiFinder provides a shared environment, it is particularly important to avoid circumventing the Window Manager.

❖ *Note:* Don't modify the visRgn field of the GrafPort in the window record; MultiFinder relies on this field.

Don't manipulate Menu Manager data structures directly

Much of MultiFinder's functionality depends on using the Apple menu in novel ways. This includes controlling the menu data structures of the Apple menu. For this reason, items should be enabled and disabled through traps provided for that purpose; direct manipulation of data structures should be avoided completely.

Don't draw on the desktop

Your application no longer "owns" the desktop under MultiFinder, so don't draw on the desktop. This means on the menu bar, desktop, or windows that belong to other applications. To remain MultiFinder-compatible with future systems, draw only in response to an update event or as part of the feedback for a user action (for example, while tracking the mouse).

❖ *Note:* DeskHook, a low-memory vector that allowed applications to draw on the desktop, is no longer called by the Window Manager.

WMgrPort and GrayRgn

WMgrPort has its visRgn set to include all active screens. Its clipRgn is initially set to "wide open" (the rectangle –32767, –32767, 32767, 32767), although Window Manager routines like ClipAbove will change it. Consider this grafPort to be read-only. The global variable GrayRgn is a region that is equal to the WMgrPort's visRgn minus the menu bar area.

You should use GrayRgn to find out the shape, size, and coordinates of the screens. You will never have to use the WMgrPort directly, and should not call GetWMgrPort under any circumstances.

Don't write directly to the screen

Drawing to the screen should only be done within windows via QuickDraw. Off-screen bitmaps should be copied to the screen via CopyBits.

Don't save the contents of windows

Don't save the bitmap contents of windows to save time when displaying dialog boxes or pop-up menus; the window you save might not be yours and it might change while it is being covered up.

Handle update events

Remember that update events are very important; applications running under MultiFinder must pay close attention to them. All applications will receive update events, not just the application currently running in the foreground. When an application receives an update event, it must update the appropriate window without doing anything else.

MultiFinder feeds update events to the application when the application makes an event call and continues to feed update events to the application until it actually processes them. Applications should respond (that is, draw) as a direct response to receiving the update event.

In general, if you are using an event call, you should be prepared to receive and respond to update events immediately. Do not defer update processing to a later time.

Use null events properly

Null events have a different meaning under MultiFinder. Originally, an application would receive a null event when no other event occurred. Under MultiFinder, however, a well-behaved application receives null events when it is in the foreground and no background task is pending.

Periodic garbage collection and similar time-consuming actions should not be performed on every null event received. Use absolute time rather than the number of events.

Support keyboard commands for editing

Support the appropriate keyboard equivalents for menu editing commands; for example, Undo (Command-Z), Cut (Command-X), Copy (Command-C), and Paste (Command-V).

Memory management

Applications that do not supply a **SIZE resource** (see Chapter 3, "MultiFinder-Aware Applications," for more information on this new resource) are launched into the default partition size of 384K. Thus, you will encounter trouble if your application requires more than this amount of memory. You may want to include a SIZE resource in your application to inform MultiFinder that you require a particular partition size.

To ensure that your application remains compatible under MultiFinder, follow these memory management guidelines.

Don't depend on the relative positions of the system and application heaps

Don't make assumptions about the memory model concerning the relative positions of the various heaps. The system heap is not necessarily adjacent to the application heap.

Free space and heap size

Heap size is not as important as the amount of free space. Use FreeMem, PurgeSpace, and MaxBlock to verify how much free space is available.

The size of your heap is given only by the bkLim field of the heap zone header. You can find this by dereferencing the ApplZone pointer in low memory (or calling the ApplicZone in either C or Pascal). For instance, in C:

```
*(unsigned long*) ApplicZone()-(unsigned long)ApplicZone()
```

Stack size

If your application has unusual stack requirements you can check the size of your stack by calling StackSize.

If you must resize your stack, call SetApplLimit immediately before or after initializing the various Toolbox managers. This will indirectly change your stack size. SetApplLimit sets the limit of the application's heap size. Only the original application zone can be expanded. See ***Inside Macintosh,*** Volume II for more information.

The application heap

The only permanent memory available to your application is your application heap and your stack. If you need to allocate additional heaps, they must exist within this area.

The A5 world

The **A5 world** consists of an application's own global variables and its private set of QuickDraw globals (both are accessed through A5), the set of low-memory globals associated with the application by MultiFinder, and the application's heap and stack. The single-Finder environment only allowed one A5 world at any given time. With MultiFinder active, each open application has an A5 world.

Most applications don't need to worry about their A5 world since MultiFinder automatically ensures that the application's A5 world is set up whenever the application is given processor time. There are, however, circumstances where some portion of an application will need to make certain that it is operating in its own A5 world. Before the specific details and guidelines for these circumstances are described, consider the three possible A5 situations that can occur under MultiFinder:

- **A5 and low memory both valid**—register A5 points to the application's globals and low memory contains the set of Toolbox and OS globals appropriate for the application (including the global CurrentA5). This is the normal situation when an application is running.
- **A5 invalid, low memory valid**—register A5 points nowhere in particular; however, low memory contains the set of Toolbox and OS globals appropriate for the application (including the global CurrentA5). This situation can sometimes occur in trap patches where the A5 register is temporarily used to store a value other than the A5 world pointer.
- **A5 and low memory both invalid**—register A5 points to another application's globals and low memory contains the wrong set of Toolbox and OS globals (including the global CurrentA5). This situation can occur for routines run at interrupt time—such as completion routines, VBL tasks, Time Manager tasks—and in application-specific window definition procedures (WDEF's).

Trap patches and global data

If you are patching traps, use the SetTrapAddress calls rather than writing into the dispatch table in low memory. To ensure that MultiFinder will only run your trap patches while your application is running in the foreground, place your patch-receiving routines in your application heap and not in the system heap.

Remove all your patched traps before exiting from your application. Under MultiFinder, patches are local to your application and no longer exist after the application quits. However, to remain compatible with the single-Finder environment, your application must remove them.

For those traps that cannot be called from interrupt routines (such as calls to QuickDraw), you cannot assume that the value of A5 points to your application's globals when a trap is made. This means that if you patch a trap and the code you install references your application's global data, you must manually save A5, set it to CurrentA5, do your work, and restore the original A5 when you are finished.

Completion routines

Many I/O completion routines (for asynchronous I/O) run at interrupt level; this implies that any A5 world could be active. Applications cannot rely on A5 or CurrentA5 to contain the correct value when the I/O completion routine is called.

Place the value of CurrentA5 that "belongs" to your partition in a place where you can find it from within your completion routine. Since it is guaranteed that A0 will be pointing to your parameter block when your completion routine is called, you can put the value of CurrentA5 at a known offset from the beginning of your parameter block and then reference it from A0. The following section on VBL tasks gives a simple example of how to do this.

VBL tasks

VBL tasks (tasks performed during the vertical retrace interrupt) in the application heap only run if the creating application is frontmost. VBL tasks in the system heap run all the time, and as in the case of interrupt routines, absolutely no A5 world context can be guaranteed.

As with I/O completion, the A5 value can be prefixed to the VBL queue element—this should be done whether the VBL tasks are in the application or the system heap.

The following short MPW examples show how to do this using INLINEs. Please note that this technique does *not* involve writing into your code segment. The value of CurrentA5 is placed in a position where the application can find it from within the VBL task. These examples rely on the fact that at the time your VBL task is run, register A0 points to the `VBLTask` structure associated with your VBL task. Since you store your CurrentA5 into the 4 bytes before the `VBLTask`, you can get the correct CurrentA5 from –4(A0).

This example also serves to demonstrate how you might write a completion routine for an asynchronous Device Manager call. It is not intended to be a complete program, nor to demonstrate optimal techniques for displaying information. In MPW™ Pascal:

```
PROGRAM InlineVBL;

USES
  {$PUSH}                         { save compiler options   }
  {$LOAD PasDump.dump}            { load symbol table dump  }
```

```
   Memtypes,QuickDraw,OSIntf,ToolIntf,PackIntf,MacPrint,WLW,JimLib;
   {$LOAD}                           { turn off LOAD                  }
   {$POP}                            { restore compiler options       }
   {$D+}                             { debug symbols                  }

CONST
   Interval    = 6;                  { how often you want your VBL    }
                                     {  called, in ticks              }
   CurrentA5   = $904;               { low-memory global              }

TYPE
   MyVBLType =   RECORD
                   CurA5: Longint; { put CurA5 where you can find it}
                   MyVBL: VBLTask; { the actual VBLTask }
                 END; {MyVBLType}

VAR
   Err         : Integer;
   MyVBLRec    : MyVBLType;
   Counter     : Integer;
   MyEvent     : EventRecord;


PROCEDURE _DataInit;
   EXTERNAL;


PROCEDURE PushA5;
   INLINE $2F0D;             { MOVE.L A5,-(SP)} {push A5 onto the stack}


PROCEDURE PopA5;
   INLINE $2A5F;             { MOVE.L (SP)+,A5} {pop the stack into A5}


PROCEDURE GetMyA5;
   INLINE $2A68,$FFFC;       { MOVE.L -4(A0),A5 }

   { Get the value of A5 you've stored before the parameter block and}
   { put it in A5. Since you know that when a VBL task is called, A0 }
   { will point to your parameter block, you also know that the value}
   { of CurrentA5 that you stored will be at -4(A0).}


PROCEDURE DoVBL;             { your VBL task }

BEGIN  { DoVBL }

{ First, make sure you have the A5 that you stored before your      }
{ parameter block. }

   PushA5;                   { push the value of A5 onto the stack    }
   GetMyA5;                  { get your A5 from right before the      }
                             {     parameter block                    }
                             { now you can access your globals:       }
```

```
    MyVBLRec.MyVBL.vblCount := Interval;  { run again to show that you can }
    Counter := Counter + 1;               {    set a global                }

    PopA5;                                { put back the original A5       }
  END; {DoVBL}

{----------------------------Main Program------------------------------}

BEGIN        {main PROGRAM}
  MaxApplZone;            { grow the heap to ApplLimit                     }
  UnloadSeg(@_DataInit);  { unload data init code before any allocations }
  InitMac;                { initialize Macintosh managers                  }
  InitWW(NIL);            { initialize WritelnWindow with default window }

  Counter := 0;           { initialize this }

  WITH MyVBLRec,MyVBL DO BEGIN

    CurA5     := LongPtr(CurrentA5)^;  { get current value of CurrentA5   }
    vblAddr   := @DoVBL;               { point to your task               }
    vblCount  := Interval;             { set up the interval where you'll }
                                       {    be called                     }
    qType     := ORD(vType);           { this is also necessary           }
    vblPhase  := 0;

  END;  {With}

  Err   :=   VInstall(@MyVBLRec.MyVBL);   { install your VBLTask }
  writeln('VInstall err = ',Err);

  REPEAT
    writeln(Counter);                            { write out counter       }
  UNTIL GetNextEvent(mDownMask,MyEvent);         { this allows a switch    }

  Err := VRemove(@MyVBLRec.MyVBL);     { you're finished, remove the task }
  writeln('VRemove err = ',Err);

  beep;                                { show the user you're finished    }

END.
```

Now for the MPW C example—first, the assembly routines:

```
CASE ON       ; for C


PushA5 PROC   EXPORT ; pushes A5 onto the stack -- BE CAREFUL NOT TO
                     ; DISTURB A0 here, since GetMyA5 relies on it
MOVE.L (SP)+,A1      ; get return address off the stack
MOVE.L A5,-(SP)      ; push A5
JMP  (A1)            ; return to caller

ENDP
```

```
PopA5 PROC     EXPORT
MOVE.L (SP)+,A1      ; get return address off the stack
MOVE.L (SP)+,A5      ; pop into A5
JMP (A1)             ; return to caller

ENDP


GetMyA5 PROC   EXPORT
MOVE.L (SP)+,A1      ; get return address off the stack
MOVE.L -4(A0),A5     ; get saved value of A5 and put it in A5
JMP          (A1)    ; return to caller

ENDP

END
```

Now the MPW C program:

```
#include  <types.h>
#include  <quickdraw.h>
#include  <resources.h>
#include  <fonts.h>
#include  <windows.h>
#include  <menus.h>
#include  <textedit.h>
#include  <events.h>
#include  <retrace.h>
#include <packages.h>

extern void PushA5();       /* MOVE.L A5,-(SP)   */  /* push A5 onto the stack  */
extern void PopA5();        /* MOVE.L (SP)+,A5   */  /* pop the stack into A5   */
extern void GetMyA5();      /* MOVE.L -4(A0),A5 */

/* Get the value of A5 you've stored before the parameter block and put it in */
/* A5.  Since you know that when a VBL task is called, A0 will point to your   */
/* parameter block, you also know that the value of CurrentA5 that you stored */
/* will be at -4(A0).                                                          */

void DoVBL();

typedef struct MyVBLType {

  long       CurA5;                  /* put CurA5 where you can find it    */
  VBLTask    MyVBL;                  /* the actual VBLTask                  */

} MyVBLType;

MyVBLType MyVBLRec;                  /* a variable of the above type        */
short  Counter;                      /* this needs to be global so the     */
                                     /*   VBL task can get to it           */
```

```
main()
{
  #define  Interval     6                /* how often you want your VBL called,  */
                                         /*    in ticks                          */
  #define  CurrentA5    0x904            /* low-memory global                    */

  WindowPtr    MyWindow;
  Rect         myWRect,rectToErase;
  OSErr        err;
  EventRecord  MyEvent;
  char         myStr[40];                /* this should be enough room */

  InitGraf(&qd.thePort);
  InitFonts();
  FlushEvents(everyEvent, 0);
  InitWindows();
  InitMenus();
  TEInit();

  SetRect(&myWRect,50,260,150,340);
  MyWindow = NewWindow(nil,&myWRect,"\pVBL",true,0, (WindowPtr)-1,false,0);

  SetPort(MyWindow);

  Counter = 0;                                   /* initialize this         */

  MyVBLRec.CurA5 = *(long *)(CurrentA5);         /* get the current value   */
                                                 /*    of CurrentA5         */
  MyVBLRec.MyVBL.vblAddr = DoVBL;                /* point to your task      */
  MyVBLRec.MyVBL.vblCount = Interval;

/* set up the interval at which you'll be called */

  MyVBLRec.MyVBL.qType = vType;                  /* this is also necessary  */
  MyVBLRec.MyVBL.vblPhase = 0;

  err = VInstall(&MyVBLRec.MyVBL);               /* install your VBLTask    */

  PenMode(patXor);                               /* so you can see the      */
                                                 /*     drawing flicker     */
  SetRect(&rectToErase,60,20,100,50);
  MoveTo(10,76);
  DrawString("\pClick to quit");

  while (!GetNextEvent(mDownMask,&MyEvent))      /* this allows a switch    */
  {
    MoveTo(20,20);                               /* draw a box              */

    LineTo(20,50);LineTo(50,50);LineTo(50,20);
    LineTo(20,20);LineTo(50,50);
    MoveTo(20,50);LineTo(50,20);MoveTo(60,43);

    EraseRect(&rectToErase);                     /* erase the last number   */
    NumToString(Counter,myStr);
    DrawString(myStr);                          /* draw the current value  */
                                                /*    of Counter           */
  }

  err = VRemove(&MyVBLRec.MyVBL);               /* you're finished, remove */
                                                /*   task                  */
  if (err != noErr) debugger();
  /* wait around until the user clicks before exiting */

  while (!Button());
  while (Button());

} /*main*/

  void DoVBL()                                  /* your VBL task           */
  {                                             /* DoVBL                   */

  /* First, make sure you have the A5 that you stored before your         */
  /*    parameter block.  */

    PushA5();                                   /* push the value of A5    */
                                                /*    onto the stack       */
    GetMyA5();                                  /* get your A5 from right  */
                                                /*  before the parameter   */
                                                /*  block                  */
    /* now you can access your globals: */

    MyVBLRec.MyVBL.vblCount = Interval;         /* to run again            */

    Counter += 1;                               /* to show you can set a   */
                                                /*  global                 */
    PopA5();                                    /* return the original A5  */

} /* END DoVBL*/
```

Time Manager tasks

Again, no A5 world context is guaranteed; however, unlike VBL tasks (and completion routines), a Time Manager task is *not* called with A0 pointing to the task block (A0 points to the task's routine instead). So, if you need to get at your application's globals from your Time Manager task, you'll have to write the value of CurrentA5 into your code segment at a time when you know that CurrentA5 is valid, and then use that value to set up A5 when your Time Manager task is called.

There may be some circumstances when your application will have to change the value in A5; just make sure that you restore A5's previous value when you are finished.

Defprocs

Since window defprocs are used by the **Layer Manager** (a new manager called only by the Window Manager), the A5 world present when the defproc is called might belong to *any* application. For example, a window frame may need to be redrawn while another application is running in the foreground (the foreground application has a window in front of another application's window and the frontmost window is moved). In this case, the Window Manager calls the WDEF to draw the frame and posts an update event for the application that owns the window to redraw the window contents. However, if the WDEF is part of your application, it may be called to draw a window frame while another application is active. Suddenly, A5 does not point to your application's globals.

Therefore, window defprocs cannot depend on A5 being valid. If your application installs a window defproc that needs to access global variables from your application, store a copy of your A5 safely by using the technique described in the previous section (VBL tasks) or place it in the refCon (reserved) field of the window.

Miscellaneous guidelines

Someday, the Macintosh will expect applications to run in the 680X0 user mode (as opposed to today's supervisor mode), so in preparation for that day avoid using any of the 680X0 privileged instructions. Also avoid making 680X0 TRAP or TRAPV calls.

All types of utilities, as well as applications, need to be aware of MultiFinder's shared environment. For example, screen savers should make sure that background processing continues.

Remember that applications should avoid direct manipulation of the Apple menu.

Low memory

The less your application accesses low memory, the better. Writing to low memory, however, is much more objectionable than reading, and should be avoided. In the long run, low memory will disappear, so try not to depend on it.

Interact with the global scrap by using the Scrap Manager whenever possible; avoid direct manipulation of the low-memory Scrap Manager data structures or the Clipboard file itself.

Try not to use the low-memory notification procedures (for example: IAZNotify, EjectNotify, DeskHook, and so on) unless absolutely necessary.

Asynchronous system calls

MultiFinder will wait until all currently active file system requests are completed before it brings another application to the foreground. This means that during any pending asynchronous file system request, MultiFinder will not allow activation of a different application.

❖ *Note:* This is not the last word on this issue. Future releases of MultiFinder will examine the compatibility of switching while asynchronous File Manager calls are still pending. Currently, in MultiFinder, Device Manager calls do not delay application switching.

Exit time restrictions

Do not assume that at exit time you can clear the screen to save code.

Do not destroy fields in system data structures, such as the window list, before you exit.

Remember to *exit gracefully*—clean up, call ExitToShell (don't assume that you can do anything you want before you call ExitToShell; even though your application is exiting, there may be other applications running), don't call InitWindows again, and so on.

System resources

As stated earlier, resources from the System file that were formerly loaded into the application heap are now loaded into the system heap for use by all applications. If a resource came from the System file, it will be loaded into the system heap even if the resSysHeap bit isn't set.

Your applications should not make assumptions about where resources other than those in your own resource files have been loaded (the system or application heap). The best way to get your own copy of a system resource is to use HandToHand rather than DetachResource.

Since other applications may need to use system resources, applications should not call ReleaseResource or DetachResource for system resources such as pointers and fonts—nor should they change resource attributes or modify the resource data directly.

Chapter 3

MultiFinder-Aware Applications

A **MultiFinder-aware application** is one that handles suspend and resume events, includes a SIZE resource, calls WaitNextEvent, and uses normally unused null events for effective background work, while not significantly affecting the responsive nature of the application currently running in the foreground. This chapter will describe each of these important aspects of the MultiFinder-aware application.

When an application stops executing in the background and begins running in the foreground, it has all the rights and responsibilities of any foreground application. These include being a good citizen while running in the foreground by calling WaitNextEvent (see information on this new event call later in this chapter) to allow other applications the opportunity to perform some work while running in the background.

After an application begins running in the foreground, it can receive user events and use any Toolbox service (File Manager, QuickDraw, Window Manager, and so on). Keep in mind that the same application executing in the background can also use any Toolbox or OS service, but won't see user events until it begins running in the foreground.

When a user attempts to bring a second application to the foreground, the Event Manager checks to see if the applications involved can handle suspend/resume events. Here, the user is trying to switch between two layers. If your application doesn't handle suspend/resume events, the operation of the Macintosh will appear sluggish.

Suspend and resume events

Two new types of **app4Evts** (type 4 application events) have been created within the Event Manager: **suspend** and **resume.** Their primary function is an optimization to tell the application when it should process the scrap. A secondary function is to tell the application whether it is in the foreground or background.

❖ *Programming tip:* It's a good idea to have a variable (for example, InForeground, initialized to TRUE) that keeps track of whether the application is in the foreground or background. When your application is launched, assume that you are in the foreground. If you receive a suspend event, you're going to the background; if you receive a resume event, you're going to the foreground.

If you intend to have your application perform work in the foreground and the background, you need to process these two events.

Table 3-1 lists the meanings of the bits in the suspend/resume event message field.

Table 3-1
The suspend/resume message field

Bit	**Meaning**
0	0 = suspend event 1 = resume event
1	0 = Clipboard conversion not required on resume 1 = Clipboard conversion required on resume
2–23	reserved
24–31	high-byte value of $01 indicates a suspend or resume event high-byte value of $FA indicates a mouse-moved event

Handling activate and deactivate

Suspend/resume events have to be handled whenever you get them; usually, this happens in the main event loop. By supporting suspend/resume events, the application is taking responsibility for activating or deactivating its front window at suspend/resume time.

If the multiFinderAware bit has been set in the application's SIZE resource (see the following section for more information on this new resource), then the application must take responsibility for performing a deactivate after receiving a suspend event and an activate after receiving a resume event.

Take care if masking out app4Evts

Suspend/resume events are not queued, so be careful when masking out app4Evts. You will still get switched out; however, all that will happen if you mask out app4Evts is that your application won't know when it is going to be switched out (your application will still be switched out when you call WaitNextEvent). If your application sets a boolean to tell whether or not it's in the foreground or the background, you definitely don't want to mask out app4Evts.

A C example of how to handle suspend and resume events

If an application doesn't support the suspend/resume events, MultiFinder has to trick the application into performing **scrap coercion** to ensure that the contents of the Clipboard can be transferred from one application to another. This process adds to the time it takes to move the foreground application to the background and vice versa and makes the user interface appear cumbersome.

An application responds to a suspend event by moving its private scrap into the Clipboard and then returning to the main event loop. When the application receives a resume event, and if the Clipboard has been altered, the application copies the Clipboard and converts it back to its private scrap. After this transformation, the application resumes executing.

❖ *Note:* Applications should hide their Clipboard window when not running in the foreground. The contents of the Clipboard window are not valid unless the application is frontmost.

MultiFinder sets bit 1 of the EventRecord of resume events if the scrap has changed while the application was suspended.

Here is a C example of how to handle the suspend/resume events (this code example also uses the new MultiFinder call **WaitNextEvent**; see the section on this call presented later in this chapter):

```
/* --- Useful macros for determining specifics of suspend/resume events ----*/

#define App4Selector(eventPtr)     (*((unsigned char *) &(eventPtr)->message))
/* top byte of message field is the selector */

#define SUSPEND_RESUME_SELECTOR          0x01
```

```
/* selector of this value is suspend/resume */

#define SuspResIsResume(evtMessage)      ((evtMessage) & 0x00000001)
/* low bit on signifies resume */

#define SuspResIsSuspend(evtMessage)     (!SuspResIsResume(evtMessage))
/* low bit off signifies suspend */

#define ScrapDataHasChanged(evtMessage) ((evtMessage) & 0x00000002)
/* only valid for suspend/resume messages */

/* ---------------------    Necessary global variables    ---------------------- */

Boolean       wneIsImplemented;    /* Is _WaitNextEvent implemented?              */

Boolean       inForeground;        /* Is this application in the foreground under */
                                   /*    MultiFinder?                             */

WindowPtr     clipboardWindow;     /* This is a pointer to the Clipboard window,  */
                                   /* which is only made invisible when the user  */
                                   /* closes it.  The next time it is opened      */
                                   /* make it visible again to speed up the       */
                                   /* process.                                    */

/* -------------------------    The main event loop    ------------------------- */

/* pick the biggest possible timeout for _WaitNextEvent */

#define BIG_TIMEOUT        0xFFFFFFFF

void
EventLoop()
{
  EventRecord       myEvent;
  void              HandleMouseDownEvent(EventRecord *pEvent);
  void              HandleKeyDownEvent(EventRecord *pEvent);
  void              HandleUpdateEvent(EventRecord *pEvent);
  void              HandleActivateEvent(EventRecord *pEvent);
  void              HandleApp4Event(EventRecord *pEvent);

  for (;;)                              /* standard method for looping forever */
  {
  /* check the following each time through loop */

    CheckClipboardWindow();

    if (wneIsImplemented)               /* get an event */
    {
      if (!WaitNextEvent(everyEvent, &myEvent, BIG_TIMEOUT, nil))
        continue;                       /* keep looping until you get a valid    */
      }                                 /*    event                              */
    else
    {
      SystemTask();                     /* the system will call this itself if   */
```

```
                                        /*   _WaitNextEvent is used              */
        if (!GetNextEvent(everyEvent, &myEvent))
          continue;
      }

      switch (myEvent.what)
      {
        case mouseDown:
                HandleMouseDownEvent(&myEvent);
                break;
        case keyDown:
                HandleKeyDownEvent(&myEvent);
                break;
        case updateEvt:
                HandleUpdateEvent(&myEvent);
                break;
        case activateEvt:
                HandleActivateEvent(&myEvent);
                break;
        case app4Evt:
                HandleApp4Event(&myEvent);
                break;
        default:
                break;
      }
    }
}

/* ----------------------------ConvertPrivateScrapToDesk------------------------- */
void
ConvertPrivateScrapToDesk()

/* If the application uses a private scrap for the Clipboard contents, then this*/
/* is the place to make it public.  This would normally be called after         */
/* receiving a suspend event (so that it gets to a location from which it can be*/
/* sent to other applications), when the application quits, or when a desk      */
/* accessory is activated.*/
   {
   }

/* ---------------------------ConvertDeskScrapToPrivate-------------------------- */
void
ConvertDeskScrapToPrivate()

/* The complement to ConvertPrivateScrapToDesk(), this converts the public      */
/* (desk) scrap to the application's private scrap, if it exists (one example   */
/* is the textedit scrap).  This would normally be called after receiving a     */
/* resume event, when the application starts up, or when a desk accessory is    */
/* deactivated. */
   {
   }

/* ------------------------------HandleApp4Event---------------------------------- */
void
```

```
HandleApp4Event(pEvent)
EventRecord         *pEvent;

/* Handle the app4Evt (as determined by pEvent->what == 15) if and only if it's */
/* a suspend/resume event.                                                       */

/* NOTE:  This code only applies if the multiFinderAware flag is set in the     */
/* application's SIZE resource.                                                  */

{
  void      MyDeactivateWindow(WindowPtr pWindow);
  void      MyActivateWindow(WindowPtr pWindow);
  void      HideClipboard();
  void      ShowClipboard();

  if  (App4Selector(pEvent)  ==  SUSPEND_RESUME_SELECTOR)
  /* if it's not suspend/resume then ignore it */
  {
    register  WindowPtr frontWindow  =  FrontWindow();
    static  Boolean clipboardVisInFG;

    /* If visible in the foreground, you have to hide it before going to        */
    /* the background, but then show it later.  This is important because the   */
    /* Clipboard contents are not valid unless the application is in the       */
    /* foreground (i.e., in the frontmost layer). */

    /* It's either suspend or resume, based on the low bit of the message field.*/
    /* You have to treat suspend as a deactivate on the front window and resume */
    /* as an activate on it because you have the multiFinderAware flag set     */
    /* in the SIZE resource.                                                     */

    if  (SuspResIsSuspend(pEvent->message))
    {
      /* --------------------------   Suspend Event   ---------------------------- */
      inForeground  =  false;
      /* suspend event signifies you are moving to the background */

      if  (ScrapDataHasChanged(pEvent->message))
      /* on a suspend, this signifies whether the user has changed the Clipboard */

        ConvertPrivateScrapToDesk();
        if (frontWindow != nil)
          MyDeactivateWindow(frontWindow);
          /* treat the suspend event as you would a deactivate event */

        if (clipboardWindow != nil &&
              ((WindowPeek)clipboardWindow)->visible)
        {
          HideClipboard();  /* hide the Clipboard when you're in the background */
          clipboardVisInFG  =  true;
        }
        else
          clipboardVisInFG  =  false;
    }
```

```
        else
        {
          /* ---------------------------   Resume Event   ---------------------------*/

          inForeground = true;
          /*     resume event signifies that you are returning to the foreground       */

          if  (ScrapDataHasChanged(pEvent->message))       /* if new scrap, then      */
                                                           /* reset your private one  */
            ConvertDeskScrapToPrivate();
          if (frontWindow != nil)
            MyActivateWindow(frontWindow);

          /* have to treat the resume event as you would an activate event*/
          if (clipboardVisInFG)
            ShowClipboard();
        }
      }
    }
```

The SIZE resource

The **SIZE resource** (see Table 3-2) is used to communicate information from the application to MultiFinder. You are responsible for creating and maintaining the information for this resource.

When an application is launched under MultiFinder, it is placed into a memory partition that cannot change in size. It is the application's responsibility to inform MultiFinder just how large a memory partition it will require.

The SIZE resource consists of a 16-bit **flags** field, used to communicate to MultiFinder the level of responsibility an application will handle, directly followed by a 32-bit **minimum size** field and a 32-bit **preferred size** field, which indicate the minimum and preferred sizes the application will operate within. The minimum size is the actual limit below which your application will not run. The preferred size is the memory size at which your application can run effectively.

Table 3-2
The SIZE resource

Bit	**Meaning**
0–8	reserved
9	getFrontClicks
10	onlyBackground
11	multiFinderAware

12	canBackground
13	reserved
14	acceptSuspendResumeEvents
15	reserved
16–48	preferred size
49–81	minimum size

The SIZE resource flags

Here are the SIZE resource flags:

- **acceptSuspendResumeEvents**—when set, this bit signifies that the application knows how to process suspend/resume events. When true, MultiFinder notifies the application before making it inactive and after reactivating it. In this way, the application knows when to process the global scrap.

  Failure to support this optimization requires MultiFinder to trick the application into performing scrap coercion to ensure that the contents of the Clipboard can be transferred from one application to another. This process adds to the time it takes to move the foreground application to the background and vice versa. MultiFinder will also create a false window to cause the foreground application's window to be deactivated unless the multiFinderAware bit is set.

  Whenever an application calls one of the event calls, MultiFinder can return a suspend event. After receiving a suspend event, an application does not actually become inactive until the next event call. At this time, the application should convert any local scrap into the global scrap and hide mini-windows, selections, and so on.

  When control returns to the application, MultiFinder returns a resume event. The application may now convert the global scrap back into its own private scrap, if necessary. As part of the resume event, MultiFinder also lets the application know if the Clipboard has changed since the application was suspended by setting bit 1 of the message field of the EventRecord of resume events.

  ❖ *Programming tip:* If you set the acceptSuspendResumeEvents bit, set the multiFinderAware bit as well.

- **canBackground**—when set, this bit means that the application wants to receive null events while in the background. If your application has nothing to do while in the background, don't set this bit.

- **multiFinderAware**—when set, this bit means that an application takes responsibility for activating and deactivating any windows in response to a suspend/resume event. This means that if the application was suspended and the acceptSuspendResumeEvents flag was set and the multiFinderAware flag was not set, then the application would still receive an activate event. If you set the multiFinderAware flag, the application won't receive activate events—you must take care of activation and deactivation yourself when you receive the corresponding suspend or resume event.

  Because you have taken responsibility for deactivation, if the application's window is on top, the suspend event should also be treated as though a deactivate event were received as well (if both the multiFinderAware and acceptSuspendResumeEvents flags were set). For example, scroll bars should be inactivated, blinking insertion points should be hidden, selected text should be deselected if your application moves to the background, and so forth. If you don't set this bit, MultiFinder has to create a window to force the activate/deactivate events to occur.

  ❖ *Remember:* Your application cannot take full advantage of the speed increases obtained from the suspend/resume events unless you set the multiFinderAware bit.

- **onlyBackground**—set this flag if your application does not have a user interface and will not run in the foreground.
- **getFrontClicks**—set this flag if you want to receive the mouse-down and mouse-up events used to bring your application to the foreground when the user clicks in one of your application's windows while it is suspended. Ordinarily, the mouse-down and mouse-up events that trigger such a switch are not sent to the application.

Preferred memory size

The preferred size is an amount of memory in which an application will run effectively, and which MultiFinder will attempt to secure upon launch of the application. If this amount of memory is unavailable, the application is placed into the largest contiguous block available providing that it is larger than the specified minimum size. Users can modify the preferred size through the Finder's Get Info window.

❖ *Note:* If the amount of available memory is between the minimum and preferred sizes, MultiFinder will display a dialog box asking if the user wants to run the application.

Minimum memory size

The minimum size is an actual limit below which the application will not run. The only way users can see the minimum size is if they try to create a partition smaller than the minimum size or open the Finder's Get Info window.

How to create your own SIZE resource

There is no simple formula for determining the appropriate size requirement for all applications. Since there are so many factors that affect memory requirements, only general guidelines are applicable.

An application's memory requirement depends on a number of factors—the static heap size, dynamic heap, A5 world, and the stack. The static heap size includes objects that are always present during the course of execution of the application (these could be code segments, Toolbox data structures for window records, and so on). Dynamic heap requirements come from various heap objects created on a per-document basis (which may vary in size proportionately with the document itself) and objects that are required for specific commands or functions. The size of the A5 world depends on the amount of global data and intersegment jumps the application contains. The stack contains variables, return addresses, and temporary information.

How much memory will an application require? Macsbug and its heap-exploring commands can be helpful in empirically determining the application's appetite for memory. Checking to see what resides in the application's heap at key times while performing all the application's functions would be quite worthwhile.

The preferred size should be chosen to allow the application to perform almost all of its functions without problems. On the other hand, the application shouldn't be too greedy. Remember that in the MultiFinder environment, multiple applications are sharing the machine.

The minimum size should be chosen such that the application would never cause a system error if required to run within that amount of memory.

The SIZE resource, specified by the application, is of type 'SIZE' with ID (–1). This will tell MultiFinder what you suggest as the preferred and minimum sizes. The user has the option of changing the application's preferred size, but not *below* the minimum size. Rather than lose the original preferred size, a second SIZE resource (SIZE, 0) is created to show the user's specified preferred size. When MultiFinder prepares to launch an application, it first checks the (SIZE, 0) resource. If this doesn't exist, MultiFinder then looks for the (SIZE, –1) resource.

Applications should not modify the preferred or minimum memory requirements of the SIZE resource; however, if this is absolutely necessary, you must change both the (SIZE, –1) and the (SIZE, 0) resources to affect the attributes mentioned above.

How can I tell if my application is running in the background?

An application can tell if it is running in the background if it has received a suspend event but not the corresponding resume event.

To run in the background under MultiFinder, an application must have set the canBackground bit (bit 12 of the FLAGS word) in the SIZE resource. In addition, the acceptSuspendResumeEvents bit (bit 14) should be set.

Null events

As stated in Chapter 2, null events have a different meaning under MultiFinder. A MultiFinder-aware application receives null events when it is in the foreground and no background task is pending, or if the application is in the background and the canBackground bit is set in the SIZE resource.

Also remember that periodic garbage collection and similar time-consuming actions should not be performed on every null event received.

WaitNextEvent

In MultiFinder, there is a new Event Manager call—**WaitNextEvent**—that will allow the system to run more efficiently. There are two important differences between WaitNextEvent and GetNextEvent. WaitNextEvent allows the caller to specify, in addition to an event record and mask, a time (sleep) during which the application relinquishes the processor if no events are pending; and it also allows the caller to specify a region (mouseRgn) from which control will not be returned until the mouse is moved outside that region.

❖ *Note:* GetNextEvent is equivalent to WaitNextEvent with a sleep value of 0 and a mouseRgn value of 0. Also, an application will now receive suspend/resume events when calling GetNextEvent.

The interface for WaitNextEvent is:

```
Function  WaitNextEvent  (eventMask        :   INTEGER;
                           VAR theEvent    :   EventRecord;
                           sleep           :   LongInt;      {  tick units   }
                           mouseRgn        :   RgnHandle )  :  BOOLEAN;
```

The mouseRgn parameter

By taking advantage of the "automatic" mouse-tracking feature (mouseRgn) of WaitNextEvent, you can considerably simplify the application's cursor tracking. The application will receive a mouse-moved event only when the mouse strays outside the specified region.

The application can compute the region where the pointer shape should remain the same. When the mouse moves outside this region, the application receives the mouse-moved event and can change the pointer, recompute the new region for this pointer, and call WaitNextEvent again. The region is given in global coordinates. If you pass an empty region or a nil region handle (0), mouse-moved events are not generated.

The sleep parameter

The sleep parameter (specified in ticks) allows an application to "sleep" until an event occurs or the specified time has elapsed. Passing a 0 in the sleep parameter for WaitNextEvent means that your application wants to regain control as soon as possible. This will still yield a minimal amount of time to other applications.

An application running in the background will not receive null events unless the canBackground bit is set in its SIZE resource. If an application needs to perform some work in the background, you can specify how often it needs to receive null events by adjusting the sleep parameter (for example, if your application only needs to receive one null event per second, set the sleep parameter to 60).

❖ *Programming tip:* Currently, MultiFinder will not suspend your application when the frontmost window is a modal dialog box with a window of type dboxProc—so if you want your application to perform work while in the background, don't display a dboxProc window.

Yielding time gracefully

In general, you should use WaitNextEvent instead of GetNextEvent. Any foreground application that uses WaitNextEvent with the appropriate sleep and mouseRgn values will give the maximum amount of time to any applications running in the background. Each application running in the background should also use WaitNextEvent as a means to sleep between successive invocations.

Because MultiFinder doesn't support preemptive scheduling, any application running in the background must call WaitNextEvent at regular intervals to retain the responsive nature of the application currently operating in the foreground. Poor response time is a sign that your application is not calling WaitNextEvent often enough while running in the background.

When an application using background time has control, user events destined for the frontmost application will not be handled until the application running in the background calls WaitNextEvent.

Using unused null event time

Only a user can activate an application to run in the foreground. Each time any application is scheduled, it runs until it makes an event call. MultiFinder schedules an application *ready* to perform work in the background when the application running in the foreground has no current events pending and no window updates are needed. As long as the application running in the background periodically calls WaitNextEvent, the foreground application will continue to get null events at regular intervals so that pointer tracking and insertion point blinking can continue.

Don't call SystemTask

If you call WaitNextEvent, MultiFinder will be responsible for giving time to drivers (that is, the system will call SystemTask). The important point here is that since applications running in the background are not guaranteed processing time and may be in a sleep state at any time, they cannot call SystemTask a sufficient number of times.

When exactly are applications moved between the foreground and the background?

Applications are moved between the foreground and background when you make an event call. If you have the acceptSuspendResumeEvents bit set in the SIZE resource, you will receive suspend/resume events. When you receive a suspend event from an event call you will be moved from the foreground to the background the next time you make an event call. When an application receives a suspend event, it is going to be switched, so don't do anything to try to retain control.

◆ *Programming tip:* Masking out the suspend event is not a good Macintosh programming technique. This is particularly important if you are setting a flag to tell if your application is in the foreground or background.

How can I tell if WaitNextEvent is implemented?

WaitNextEvent is part of Finder version 6.0. Most applications should *not* need to know if MultiFinder is running since future systems might include WaitNextEvent whether or not MultiFinder is running. Most of the time, the application really needs to know something like: "How can I tell if WaitNextEvent is implemented?"

The following Pascal and C code fragments are included here to demonstrate how to check whether WaitNextEvent is implemented (this code compares the trap for WaitNextEvent with the unimplemented trap). Common to both of these code examples is a useful routine, called `TrapAvailable`, to check if a particular trap is available. Here is the Pascal code for `TrapAvailable`:

```
FUNCTION TrapAvailable(tNumber: INTEGER; tType: TrapType): BOOLEAN;

CONST
  UnimplementedTrapNumber = $A89F;            {trap number of "unimplemented trap"}

BEGIN {TrapAvailable}

{Check and see if the trap exists.  On 64K ROM machines, tType will be ignored.}

  TrapAvailable := ( NGetTrapAddress(tNumber, tType) <>
                          GetTrapAddress(UnimplementedTrapNumber) );

END;  {TrapAvailable}
```

Here is the C code for `TrapAvailable`:

```
Boolean
TrapAvailable(tNumber, tType)
short    tNumber
TrapType tType
{                                  /* define trap number for old MPW or non-MPW C */

#ifndef _Unimplemented
#define _Unimplemented 0xA89F
#endif

/* Check and see if the trap exists.  On 64K ROM machines, tType will be ignored. */

      return( NGetTrapAddress(tNumber, tType) !=  GetTrapAddress(_Unimplemented) );
}
```

Here's the Pascal code segment that shows how you should set up the call to the function that will actually check to see if WaitNextEvent is implemented, followed by the skeleton for calling either WaitNextEvent or GetNextEvent and SystemTask—depending on the outcome:

```
{Note that you call both GetNextEvent and SystemTask if WaitNextEvent isn't
available.}

...
    hasWNE := WNEIsImplemented;
...
    IF hasWNE THEN BEGIN                { call WaitNextEvent }
```

```
      ...
    END ELSE BEGIN                        { call SystemTask and GetNextEvent       }
      ...
    END;
...
```

Here's the Pascal code segment that checks to see if WaitNextEvent is implemented:

```
FUNCTION WNEIsImplemented: BOOLEAN;

CONST
  WNETrapNumber = $A860;                {trap number of WaitNextEvent            }

VAR
  theWorld       : SysEnvRec;           {used to check if machine has new traps }
  discardError   : OSErr;               {used to ignore OSErr return from        }
                                        (    SysEnvirons                         }
BEGIN {WNEIsImplemented}

{ Since WaitNextEvent and HFSDispatch both have the same trap number ($60), you }
{ can only call TrapAvailable for WaitNextEvent if you are on a machine that    }
{ supports separate OS and Toolbox trap tables. Here, call SysEnvirons and      }
{ check if machineType < 0.}

  discardError := SysEnvirons(1, theWorld);

      { Even if you get an error from SysEnvirons, the SysEnvirons glue has set }
      { up machineType.}

  IF theWorld.machineType < 0 THEN
            WNEIsImplemented := FALSE   { this ROM doesn't have separate trap    }
                                        { tables or WaitNextEvent                }
  ELSE                                  { check for WaitNextEvent                }

    WNEIsImplemented := TrapAvailable(WNETrapNumber, ToolTrap);

END;  { WNEIsImplemented }
```

Here's the same example in C:

```
/* Note that you call both GetNextEvent and SystemTask if WaitNextEvent isn't    */
/* available. */

...
    hasWNE = WNEIsImplemented();
...
    if (hasWNE) {                       /* call WaitNextEvent                     */
      ...
    }
    else {                              /* call SystemTask and GetNextEvent       */
      ...
    }
...
Boolean
WNEIsImplemented()
{
/* define trap number for old MPW or non-MPW C */

#ifndef _WaitNextEvent
#define _WaitNextEvent 0xA860
#endif

  SysEnvRec theWorld;                    /* used to check if machine has new traps */

/* Since WaitNextEvent and HFSDispatch both have the same trap number ($60), you  */
/* can only call TrapAvailable for WaitNextEvent if you are on a machine that     */
/* supports separate OS and Toolbox trap tables. Call SysEnvirons and check if    */
/* machineType < 0.                                                               */

  SysEnvirons(1, &theWorld);

/* Even if you get an error from SysEnvirons, the SysEnvirons glue has set up    */
/* machineType. */

  if (theWorld.machineType < 0) {
    return(false) /* this ROM doesn't have separate trap tables or WaitNextEvent  */
  }
  else {
    return(TrapAvailable(_WaitNextEvent, ToolTrap));   /* check for WaitNextEvent  */
  }
}
```

❖ *Note:* WaitNextEvent does not conflict with any OS trap, so the above test is valid on 64K ROMs.

Temporary memory allocation calls

To reduce the memory requirements of an application's heap, MultiFinder provides a set of temporary memory allocation services that can be used for large *transient* memory requirements.

An application now has the option of using MultiFinder's temporary memory allocation calls to get additional memory; however, don't rely on always getting it because this additional memory may not be available. The application should still work if there is no additional memory available when you need it. Also, this memory is meant to be transitory; the application should use the memory for a limited time and then return it to the system for other applications to use.

This temporary memory should be released before you call WaitNextEvent again. Make the call, use the memory, and then release it.

It is important to remember a number of things when using this memory. First, you must use the temporary memory allocation calls when referencing these relocatable blocks because of the different handle requirements. Second, be sure to release the blocks of memory as soon as possible to allow other applications to use them, and to allow the user to launch new applications. Finally, never structure your application such that it depends on the availability of any of this temporary memory. This means having a backup plan in place should no temporary memory be available (most likely reserving an emergency amount of memory within your heap zone to complete the common procedures).

❖ *Note:* Do not treat these calls as Memory Manager blocks. For example, don't call GetHandleSize or SetHandleSize. Also, don't call Toolbox routines that will call GetHandleSize or SetHandleSize.

For example, the Finder now uses these temporary memory calls to secure copy buffer space to be used during file copy operations. Any available memory (unused by running applications) is dedicated to this purpose. The Finder releases the memory as soon as the copy is completed, thus making the memory available again to other applications, or to MultiFinder for launching new applications.

If the Finder cannot allocate this large temporary copy buffer, it will perform the copy using a reserved small copy buffer from within its own heap zone. This is clearly more desirable than refusing to copy (or worse yet, crashing) because no temporary memory was available.

There are several temporary memory allocation calls:

■ `FUNCTION MFFreeMem :LONGINT`

  MFFreeMem returns the total amount of free memory available for temporary allocation, in bytes.

■ `FUNCTION MFMaxMem(VAR grow:Size) : Size`

  MFMaxMem compacts the MultiFinder heap zone and returns the number of bytes of the largest contiguous free block for temporary allocation.

■ `FUNCTION MFTempNewHandle(logicalSize:Size;VAR resultCode:OSErr):Handle`

  MFTempNewHandle attempts to allocate a new relocatable block of logicalSize bytes for temporary usage and return a handle to it. The new block will be unlocked and unpurgeable. If an error occurs, MFTempNewHandle will return nil.

Result codes:	noErr	No error
	memFullErr	Not enough room

■ `FUNCTION MFTopMem: Ptr`

  MFTopMem returns a pointer to the top of the addressable RAM space.

  ❖ *Note:* Do not use this call to calculate the size of your application's memory partition. This call provides the total amount of useable machine memory—not the amount of memory available to your application.

■ `PROCEDURE MFTempDisposHandle(h:Handle; VAR resultCode:OSErr)`

MFTempDisposHandle releases the memory occupied by the relocatable block whose handle is h.

Result codes:		
	noErr	No error
	memWZErr	Attempt to operate on a free block

■ `PROCEDURE MFTempHLock(h:Handle; VAR resultCode:OSErr)`

MFTempHLock locks the specified relocatable block, preventing it from being moved within the MultiFinder heap zone.

Result codes:		
	noErr	No error
	nilHandleErr	Nil master pointer
	memWZErr	Attempt to operate on a free block

■ `PROCEDURE MFTempHUnlock(h:Handle; VAR resultCode:OSErr)`

MFTempHUnlock unlocks the specified relocatable block, allowing it to move.

Result codes:		
	noErr	No error
	nilHandleErr	Nil master pointer
	memWZErr	Attempt to operate on a free block

How can I tell if the temporary memory allocation calls are implemented?

The technique that's used to determine this is similar to the technique for determining if WaitNextEvent is implemented. In Pascal:

```
FUNCTION TempMemCallsAvailable: BOOLEAN;

CONST
   OSDispatchTrapNumber = $A88F;  { trap number of temporary memory calls }

BEGIN { TempMemCallsAvailable }

{ Since OSDispatch has a trap number that was always defined to be a Toolbox    }
{ trap ($8F), you can always call TrapAvailable. If you are on a machine that   }
{ does not have separate OS and Toolbox trap tables, you'll still get the right }
{ trap address.                                                                  }

   { check for OSDispatch }

   TempMemCallsAvailable := TrapAvailable(OSDispatchTrapNumber, ToolTrap);

END;  {TempMemCallsAvailable}
```

Now, the same example in C:

```
Boolean
TempMemCallsAvailable()
{

/* define trap number for old MPW or non-MPW C */

#ifndef _OSDispatch
#define _OSDispatch 0xA88F
#endif

/* Since OSDispatch has a trap number that was always defined to be a Toolbox    */
/* trap ($8F), you can always call TrapAvailable. If you are on a machine that   */
/* does not have separate OS and Toolbox trap tables, you'll still get the       */
/* right trap address.                                                           */

   return(TrapAvailable(_OSDispatch, ToolTrap));      /* check for OSDispatch    */

}
```

Launching and sublaunching

Certain types of applications, such as development systems, need to launch other applications. MultiFinder provides a new platform for applications to interactively communicate with such applications. The application launched by your application will become the foreground application. A **sublaunch** is the mechanism for allowing your application to call another application. Unlike the single-Finder environment, under MultiFinder when the user quits the application that you sublaunched, control does not necessarily return to your application, but rather to the next frontmost layer.

To launch another application and keep your currently active application open, set both high bits of LaunchFlags, that is

```
LaunchFlags:= $C0000000;
```

Here is the Launch parameter block description:

```
typedef struct LaunchBlock {

   StringPtr          name;
   unsigned short     soundBuffers;
   unsigned short     launchBlockID;

#define EXTENDED_BLOCK_ID  ((unsigned short)'LC')
       unsigned long        extendedBlockLen;

#define     IS_EXTENDED_BLOCK(pLaunchBlock)
((pLaunchBlock)->launchBlockID == EXTENDED_BLOCK_ID \
&& (pLaunchBlock)->extendedBlockLen >= 4)
       unsigned short       finderFileFlags;
#define     FINDER_FILE_FLAG_MULTILAUNCH ((short)(1<<6))
       unsigned short        launchFlags;

#define     LAUNCH_FLAG_SUBLAUNCH          ((short)(1<<15))
#define     LAUNCH_FLAG_TWITCHLAUNCH              ((short)(1<<14))
#define     IS_TWITCH_LAUNCH(pLaunchBlock)  (IS_EXTENDED_BLOCK(pLaunchBlock)  \
              && (pLaunchBlock->launchFlags & LAUNCH_FLAG_TWITCHLAUNCH))
} LaunchBlock;
```

Unlike the single-Finder model, if you set both high bits of LaunchFlags, your application will continue to execute after calling Launch, so be prepared. Calling Launch with both high bits of LaunchFlags set can be thought of as a request to launch an application. The actual execution of that application's code (and hence ***suspend*** of your application) won't happen in the Launch trap, but at a later time (after a call or two to WaitNextEvent).

◆ *Warning:* The interface to the Launch trap will eventually change. Unless you are implementing an integrated development system, your application should not launch other applications.

Launch under MultiFinder will currently return an error if there isn't enough memory to launch the desired application, if the desired application can't be located, or if the desired application is already open. In the latter case, that application will not be made active.

If you sublaunched, control will return to your application; if not, your application will be terminated and the next frontmost layer will become active. If you didn't sublaunch and an error occurred, MultiFinder will do a SysBeep since your application will be terminated. If you sublaunched and an error occurred, MultiFinder will not beep and your application will have to report the error to the user.

Launch returns the error in register D0 if you are sublaunching. You can check for D0 < 0 after the sublaunch to see if the launch failed. If D0 >= 0, then the application will be launched. The following Pascal code segment will return an error if launch fails:

```
FUNCTION  LaunchIt(pLnch: pLaunchStruct):OSErr;
   INLINE      $205F, {MOVE.L (SP)+,A0 ;pointer to parameters in A0}
               $A9F2, {_Launch}
               $3E80; {MOVE.W D0,(A7) ;get function result from D0}
```

Working directories

A new Working Directory Control Block (WDCB) must be created and set as the current directory when your application is run under MultiFinder (unless the current application represents the root or exists on an MFS volume).

Under MultiFinder, when you call PBOpenWD, the ioWDProcID that you pass in is ignored. MultiFinder overrides your ioWDProcID with a unique process ID for your application so that it can deallocate all working directories that you allocated when your application terminates. Thus, you cannot use the ioWDProcID to identify your working directories when running under MultiFinder.

Therefore, whenever you open a working directory with PBOpenWD, you should pass your application's signature as the ioWDProcID and close the working directory as soon as possible with PBCloseWD. Also, remember to deallocate each WDCB, since the sublaunching process is recursive and there is a limit to the number of WDCB's that can be created. A good programming practice is to check for errors after calling PBOpenWD. A tMWDOErr (–121) error indicates that all available WDCB's have been allocated.

Chapter 4

Special-Purpose Applications

Three special types of applications are described in this chapter: embedded services, faceless background tasks, and desk accessories.

Embedded services and faceless background tasks are applications that perform almost all their work in the background. Desk accessories represent another special type of application that must follow certain programming guidelines to remain compatible with MultiFinder.

Embedded services

An **embedded service** is a special-purpose application that runs only in the background. This type of application is normally not visible and interacts heavily with the Notification Manager (see Appendix D for a detailed description).

A good example of an embedded service that prints and uses heavy amounts of background time is PrintMonitor.

PrintMonitor allows the user to interactively monitor what is being printed. While PrintMonitor is running in the background, a user can bring it to the foreground to see which jobs are being held in the print spooler, alter the document printing order, cancel or suspend any or all waiting documents, or set times for particular documents to be printed. This allows the user to print out large documents during times when the LaserWriter® might be idle or rarely used.

Faceless background tasks

A **faceless background task** is almost invisible. It is minimal in size and has no user interface—no icon will appear in the Apple menu, no windows will be displayed, and no port exists. If any user interaction is required, it uses the Notification Manager.

A faceless background task sets the canBackground and backgroundOnly bits in the SIZE resource and should not significantly affect the responsive nature of the application running in the foreground.

An example of a faceless background task is Backgrounder, a continually active but user-transparent program. Its main function in life is to seek out and identify the creation of a printer spool file. Spool files are created under MultiFinder when a user wants to print a document in the background. When Backgrounder sees a spool file, it sets the printing process in motion by launching PrintMonitor.

Desk accessories

Desk accessories were originally designed for the Macintosh environment because they offered two distinct advantages over applications. First, they incorporated a limited degree of multitasking. Second, by using the Clipboard, they offered a primitive type of interprocess communication. MultiFinder now makes these advantages available to applications as well as desk accessories.

Since MultiFinder will eventually represent the sole Macintosh desktop environment, you will be better served in the future if you design and write a small application rather than a desk accessory. This does not mean that desk accessories are not compatible with MultiFinder. While small applications are now preferable to desk accessories, MultiFinder does support the standard desk accessory model; however, there have been some changes.

The major change is that desk accessories in the System file are now loaded into the system heap rather than the application heap (except when the Option key is held down). This means that certain desk accessories that rely on being opened in the application heap of specific applications may not work under MultiFinder. Also, when a user opens a desk accessory or clicks on one already open, MultiFinder brings all open desk accessories to the foreground.

Desk accessories can be divided into two different categories: self-sufficient and dependent. Self sufficient desk accessories will continue to work as intended under MultiFinder.

Self-sufficient desk accessories

Self-sufficient desk accessories do not rely on the presence of specific applications to function—that is, they don't need to be in a particular application's heap in order to work correctly. The standard desk accessories from Apple, such as the Scrapbook and Notepad, are examples of self-sufficient desk accessories. A self-sufficient desk accessory also doesn't care about the rest of the world while it's running. Under MultiFinder, a desk accessory has no way of knowing which application was active when the user opened it.

Dependent desk accessories

A dependent desk accessory relies on an information exchange with a specific application that allows it to perform its particular function. However, under MultiFinder, this exchange breaks down because in general, the desk accessory does not load into the application heap and has no way of determining which application opened it.

An example of a dependent desk accessory is a spelling checker that only works with certain word processing applications. This sort of desk accessory won't work under MultiFinder. Such desk accessories usually use the scrap to keep the text they're going to check and rely on posting events to tell the word processing application to save the text to the scrap. Unfortunately, at accRun time, the desk accessory doesn't know which MultiFinder partition (specific application heap) called it. This means that spelling checker desk accessories can no longer post events to begin the text retrieval process.

Error checking

While it is true that MultiFinder will enlarge the system heap to make room for desk accessories if possible, all desk accessories need to contain thorough error checking to see if they have enough memory to load.

A desk accessory will not have any indication that MultiFinder has loaded it, or that there is additional room in the system heap. To prevent possible memory problems, desk accessories can check to see if there is enough memory to load by trying to allocate all the memory they need and exiting gracefully if there is not enough available.

Appendix A

A C Example of a MultiFinder-Aware Application

The following C program is an example of a MultiFinder-aware application.

```
/* ------------------------------------------------------------------------*/
/*                                                                         */
/*    MultiFinder-Aware Sample Application                                 */
/*                                                                         */
/*    Copyright © 1988 Apple Computer, Inc.                                */
/*    All rights reserved.                                                 */
/*                                                                         */
/*    This sample application was written by Macintosh Developer Technical */
/*    Support.  It displays a single, fixed-size window in which the user  */
/*    can enter and edit text.                                             */
/*                                                                         */
/*-------------------------------------------------------------------------*/


/* Segmentation strategy:

   This program consists of three segments.  Main contains most of the code,
   including the MPW libraries, and the main program.  Initialize contains
   code that is only used once, during startup, and can be unloaded after the
   program starts.  %A5Init is automatically created by the Linker to initialize
   globals for the MPW libraries and is unloaded right away. */


/* SetPort strategy:

   Toolbox routines do not change the current port.  However, this program uses
   a strategy of calling SetPort whenever you want to draw or make calls that
   depend on the current port.  This makes you less vulnerable to bugs in other
   software that might alter the current port (such as the bug (feature?) in
   many desk accessories that change the port on OpenDeskAcc).   This strategy
```

```
    also makes the routines from this program more self-contained,
    since they don't depend on the current port setting. */


/* Clipboard strategy:

    This program does not maintain a private scrap.  Whenever a cut, copy, or paste
    occurs, you import/export from the public scrap to TextEdit's scrap right away,
    using the TEToScrap and TEFromScrap routines.  If you did use a private scrap,
    the import/export would be in the activate/deactivate event and suspend/resume
    event routines. */


#include <Values.h>
#include <Types.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Controls.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <Desk.h>
#include <Scrap.h>
#include <ToolUtils.h>
#include <Memory.h>
#include <SegLoad.h>
#include <Files.h>
#include <OSUtils.h>
#include <Traps.h>                /* MPW™ 2.0.2 Traps.h is missing an #endif     */


/* MaxOpenDocuments is used to determine whether a new document can be opened   */
/*  or created.  You keep track of the number of open documents, and disable the*/
/*  menu items that create a new document when the maximum is reached.  If the  */
/*  number of documents falls below the maximum, the items are enabled again.   */

#define      maxOpenDocuments          1

/* SysEnvironsVersion is passed to SysEnvirons to tell it which version of the  */
/*    SysEnvRec is understood.                                                   */

#define      sysEnvironsVersion        1

/* OSEvent is the event number of the suspend/resume and mouse-moved events     */
/*  sent by MultiFinder. Once you determine that an event is an osEvent,         */
/*  look at the high byte of the message sent with the event to determine        */
/*  which kind of osEvent it is. To differentiate suspend and resume events,     */
/*  check the resumeMask bit.                                                    */

#define      osEvent                   app4Evt        /* event used by           */
                                                      /*   MultiFinder           */
#define      suspendResumeMessage      1              /* high byte of suspend/   */
```

```
                                                          /* resume event message    */
#define       resumeMask                1                /* bit of message field    */
                                                          /* for resume vs. suspend */
#define       mouseMovedMessage         0xFA             /* high byte of mouse-     */
                                                          /* moved event message     */

/* The following constants are all resource IDs. They correspond to resources    */
/*  in Sample.r.  See Appendix C.                                                 */

#define       rMenuBar                  128              /* application's menu bar  */
#define       rAboutAlert               128              /* about alert             */
#define       rDocWindow                128              /* application's window    */

/* The following constants are used to identify menus and their items. The menu */
/*  constants are menu IDs, and the individual item constants are item numbers   */
/*  within the menus.                                                             */

#define       mApple                    128              /* Apple menu */
#define       iAbout                    1

#define       mFile                     129              /* File menu  */
#define       iNew                      1
#define       iClose                    4
#define       iQuit                     12

#define       mEdit                     130              /* Edit menu  */
#define       iUndo                     1
#define       iCut                      3
#define       iCopy                     4
#define       iPaste                    5
#define       iClear                    6


/* A DocumentRecord contains the WindowRecord for one of the document windows,  */
/*   as well as the TEHandle for the text being edited. Other document fields   */
/*   can be added to this record as needed. For a similar example, see how the  */
/*   Window Manager and Dialog Manager add fields after the grafPort.           */

typedef struct {
   WindowRecord     window;
   TEHandle         te;
} DocumentRecord, *DocumentPeek;


/* GMac is used to hold the result of a SysEnvirons call.  This makes            */
/*   it convenient for any routine to check the environment.                     */

SysEnvRec     gMac;                     /* set up by Initialize                  */

/* GHasWaitNextEvent is set at startup, and tells whether the WaitNextEvent      */
/*   trap is available.  If it is false, GetNextEvent must be called.            */

Boolean       gHasWaitNextEvent;        /* set up by Initialize                  */
```

```
/* GInBackground is maintained by the osEvent handling routines.  Any part of   */
/*    the program can check it to find out if it is currently in the background. */

Boolean       gInBackground;             /* maintained by Initialize and DoEvent */

/* GNumDocuments is used to keep track of how many open documents there are at   */
/*   any time.  It is maintained by the routines that open and close documents.  */

short         gNumDocuments;             /* maintained by Initialize, DoNew, and */
                                         /*     DoCloseWindow                    */


/* Here are declarations for all the C routines.  In MPW 3.0 you can use         */
/*    actual prototypes for parameter type checking.                             */

void EventLoop();
void DoEvent( /* EventRecord *event */ );
void AdjustCursor( /* Point mouse, RgnHandle region */ );
void DoUpdate( /* WindowPtr window */ );
void DoDeactivate( /* WindowPtr window */ );
void DoActivate( /* WindowPtr window */ );
void DoContentClick( /* WindowPtr window, EventRecord *event */ );
void DoKeyDown( /* EventRecord *event */ );
long GetSleep();
void DoIdle();
void DrawWindow( /* WindowPtr window */ );
void AdjustMenus();
void DoMenuCommand( /* long menuResult */ );
void DoNew();
void DoCloseWindow( /* WindowPtr window */ );
void DoCloseBehind( /* WindowPtr window */ );
void Terminate();
void Initialize();
Boolean IsAppWindow( /* WindowPtr window */ );
Boolean IsDAWindow( /* WindowPtr window */ );
Boolean TrapAvailable( /* short tNumber, TrapType tType */ );


/* Define HiWrd and LoWrd macros for efficiency. */

#define HiWrd(aLong) (((aLong) >> 16) & 0xFFFF)
#define LoWrd(aLong) ((aLong) & 0xFFFF)

/* Define TopLeft and BotRight macros for convenience.  Notice the implicit     */
/*    dependency on the ordering of fields within a Rect.                       */

#define TopLeft(aRect)     (* (Point *) &(aRect).top)
#define BotRight(aRect)    (* (Point *) &(aRect).bottom)


extern void _DataInit();

/* This routine is automatically generated by the MPW Linker. This external     */
/*    reference to it is made so that its segment, %A5Init, can be unloaded.    */
```

```
#define __SEG__ Main
main()
{
    UnloadSeg((Ptr) _DataInit);          /* NOTE: _DataInit must not be in Main  */
    MaxApplZone();                       /* expand the heap so code segments     */
                                         /*     load at the top                  */

    Initialize();                        /* initialize the program               */
    UnloadSeg((Ptr) Initialize);         /* NOTE: Initialize must not be in Main */

    EventLoop();                         /* call the main event loop             */
}


/* Get events forever, and handle them by calling DoEvent.  Also call     */
/*    AdjustCursor each time through the loop.                            */

#define __SEG__ Main

void EventLoop()
{
    RgnHandle cursorRgn;
    Boolean   ignoreResult;
    EventRecord     event;

    cursorRgn = NewRgn();
    do {
       if ( gHasWaitNextEvent )
            ignoreResult = WaitNextEvent(everyEvent, &event,
                                GetSleep(), cursorRgn);
       else {
            SystemTask();
            ignoreResult = GetNextEvent(everyEvent, &event);
       }
       AdjustCursor(event.where, cursorRgn);
       DoEvent(&event);
    } while ( true );      /* loop forever */
} /*EventLoop*/


/* Do the right thing for an event. Determine what kind of event it is, and call*/
/*    the appropriate routines.                                                 */

#define __SEG__ Main
void DoEvent(event)
    EventRecord     *event;
{
    short     part;
    WindowPtr window;
    char      key;

    switch ( event->what ) {
```

```
case nullEvent:
       DoIdle();
       break;
case mouseDown:
       part = FINDWINDOW(event->where, &window);
       switch ( part ) {
              case inMenuBar:
                     AdjustMenus();
                     DoMenuCommand(MENUSELECT(event->where));
                     break;
              case inSysWindow:
                     SystemClick(event, window);
                     break;
              case inContent:
                     if ( window != FrontWindow() ) {
                            SelectWindow(window);
                            /*DoEvent(event);*/         /* use this line for   */
                                                        /*  "do first click"   */
                     } else
                            DoContentClick(window, event);
                     break;
              case inDrag:
                     DRAGWINDOW(window, event->where,
                                   &qd.screenBits.bounds);
                     break;
              case inGoAway:
                     if ( TRACKGOAWAY(window, event->where) )
                            DoCloseWindow(window);
                     break;
              }
              break;
case keyDown:
case autoKey:
       key = event->message & charCodeMask;
       if ( (event->modifiers & cmdKey) != 0 ) {
                     /* Command key down */
              if ( event->what == keyDown ) {
                     AdjustMenus();              /* enable/disable/check    */
                                                 /*    menu items properly  */
                     DoMenuCommand(MenuKey(key));
              }
       } else
              DoKeyDown(event);
       break;
case activateEvt:
       window = (WindowPtr) event->message;
       if ( (event->modifiers & activeFlag) != 0 )
              DoActivate(window);
       else
              DoDeactivate(window);
       break;
case updateEvt:
       DoUpdate((WindowPtr) event->message);
       break;
```

```
        case osEvent:
               switch (event->message >> 24) {          /* high byte of message    */
                      case mouseMovedMessage:
                             DoIdle();                  /* mouse moved is also an  */
                                                        /* idle event              */
                             break;
                      case suspendResumeMessage:
                             window = FrontWindow();
                             if ( event->message & resumeMask ) {
                                    gInBackground = false;
                                    DoActivate(window);        /* Have to treat    */
                                                               /* suspend/resume   */
                                                               /* as deactivate/   */
                                                               /* activate as well.*/
                             } else {
                                    gInBackground = true;
                                    DoDeactivate(window);
                             }
                             break;
                      }
                      break;
        }
} /*DoEvent*/


/* Change the cursor's shape, depending on its position.  This also calculates  */
/*    a region that includes the cursor for WaitNextEvent.                      */

#define __SEG__ Main
void AdjustCursor(mouse,region)
    Point     mouse;
    RgnHandle region;
{
    WindowPtr frontmost;
    RgnHandle arrowRgn;
    RgnHandle iBeamRgn;
    Rect      iBeamRect;

    frontmost  = FrontWindow();                  /* only adjust the cursor when   */
                                                 /*   you are in front            */
    if ( (! gInBackground) && (! IsDAWindow(frontmost)) ) {
            /* calculate regions for different cursor shapes */
       arrowRgn  = NewRgn();
       iBeamRgn  = NewRgn();

       /* start arrowRgn wide open */
       SetRectRgn(arrowRgn, -32768, -32768, 32767, 32767);

       /* calculate iBeamRgn */
       if ( IsAppWindow(frontmost) ) {
            iBeamRect  = (*((DocumentPeek) frontmost)->te)->viewRect;
            SetPort(frontmost);                  /* make a global version of the  */
                                                 /*      viewRect                 */
```

```
			LocalToGlobal(&TopLeft(iBeamRect));
			LocalToGlobal(&BotRight(iBeamRect));
			RectRgn(iBeamRgn, &iBeamRect);
		}

		/* subtract other regions from arrowRgn */
		DiffRgn(arrowRgn, iBeamRgn, arrowRgn);

		/* change the cursor and the region parameter */
		if ( PTINRGN(mouse, iBeamRgn) ) {
			SetCursor(*GetCursor(iBeamCursor));
			CopyRgn(iBeamRgn, region);
		} else {
			SetCursor(&qd.arrow);
			CopyRgn(arrowRgn, region);
		}

		/* get rid of local regions */
		DisposeRgn(arrowRgn);
		DisposeRgn(iBeamRgn);
	}
} /*AdjustCursor*/


/* This is called when an update event is received for a window.  It calls      */
/*    DrawWindow to draw the contents of an application window.                  */

#define __SEG__ Main
void DoUpdate(window)
    WindowPtr window;
{
    if ( IsAppWindow(window) ) {
       BeginUpdate(window);                      /* this sets up the visRgn        */
       if ( ! EmptyRgn(window->visRgn) )         /* draw if updating is needed     */
             DrawWindow(window);
       EndUpdate(window);
    }
} /*DoUpdate*/


/* This is called when a window is deactivated. */

#define __SEG__ Main
void DoDeactivate(window)
    WindowPtr window;
{
    if ( IsAppWindow(window) )
       TEDeactivate(((DocumentPeek) window)->te);
} /*DoDeactivate*/


/* This is called when a window is activated. */

#define __SEG__ Main
```

```
void DoActivate(window)
    WindowPtr window;
{
    if ( IsAppWindow(window) )
       TEActivate(((DocumentPeek) window)->te);
} /*DoActivate*/


/* This is called when a mouseDown occurs in the content of a window. */

#define __SEG__ Main
void DoContentClick(window,event)
    WindowPtr        window;
    EventRecord      *event;
{
    Point            mouse;
    Boolean          shiftDown;

    if ( IsAppWindow(window) ) {
       SetPort(window);
       mouse = event->where;                    /* get the click position         */
       GlobalToLocal(&mouse);                   /* convert to local coordinates   */

       /* extend if Shift is down */
       shiftDown = (event->modifiers & shiftKey) != 0;
       TECLICK(mouse, shiftDown, ((DocumentPeek) window)->te);
    }
} /*DoContentClick*/


/* This is called for any keyDown or autoKey events, except when the Command    */
/*   key is held down.  It looks at the frontmost window to decide what to do   */
/*   with the key pressed.                                                      */

#define __SEG__ Main
void DoKeyDown(event)
    EventRecord      *event;
{
    WindowPtr        frontmost;
    char             key;

    frontmost = FrontWindow();
    if ( IsAppWindow(frontmost) ) {
       key = event->message & charCodeMask;
       TEKey(key, ((DocumentPeek) frontmost)->te);
    }
} /*DoKeyDown*/


/* Calculate a sleep value for WaitNextEvent. This takes into account the things*/
/*   that DoIdle does with idle time.                                           */

#define __SEG__ Main
long GetSleep()
```

```
{
    long      sleep;
    WindowPtr frontmost;
    TEHandle  te;

    sleep = MAXLONG;                            /* default value for sleep       */
    if ( ! gInBackground ) {                    /* if you are in front and the.. */
       frontmost = FrontWindow();               /*     front window is yours...  */
       if ( IsAppWindow(frontmost) ) {
             te = ((DocumentPeek) (frontmost))->te;        /* and the           */
                                                          /*  selection is     */
                                                          /*  an insertion     */
                                                          /*  point            */
             if ( (*te)->selStart == (*te)->selEnd )
                  sleep = GetCaretTime();                 /* you need to make */
                                                          /*  the insertion    */
       }                                                  /*  point blink      */
    }
    return sleep;
} /*GetSleep*/


/* This is called whenever you get a null event or a mouse-moved event.  It     */
/* takes care of necessary periodic actions. For this program, it calls TEIdle. */

#define __SEG__ Main
void DoIdle()
{
    WindowPtr frontmost;

    frontmost = FrontWindow();
    if ( IsAppWindow(frontmost) )
       TEIdle(((DocumentPeek) frontmost)->te);
} /*DoIdle*/


/* Draw the contents of an application window. */

#define __SEG__ Main
void DrawWindow(window)
    WindowPtr window;
{
    SetPort(window);
    TEUpdate(&window->portRect, ((DocumentPeek) window)->te);
} /*DrawWindow*/


/* Enable and disable menus based on the current state.  This is called just    */
/*  before MenuSelect or MenuKey, so it can set up everything for the Menu      */
/*  Manager. Since these are the times that the user can see the menus or choose*/
/*  a menu item, you only need to enable/disable items then.                    */

#define __SEG__ Main
void AdjustMenus()
```

```
{
   WindowPtr        frontmost;
   MenuHandle       menu;
   Boolean          undo;
   Boolean          cutCopyClear;
   Boolean          paste;
   TEHandle         te;

   frontmost = FrontWindow();

   menu = GetMHandle(mFile);
   if ( gNumDocuments < maxOpenDocuments )
      EnableItem(menu, iNew);                   /* New is enabled when you can    */
                                                /*     open more documents        */
   else
      DisableItem(menu, iNew);
   if ( frontmost != nil )                      /* Close is enabled when there    */
                                                /*    is a window to close        */
      EnableItem(menu, iClose);
   else
      DisableItem(menu, iClose);

   menu = GetMHandle(mEdit);
   undo = false;
   cutCopyClear = false;
   paste = false;

   if ( IsDAWindow(frontmost) ) {
      undo = true;                              /* all editing is enabled for     */
                                                /*    DA windows                  */
      cutCopyClear = true;
      paste = true;
   } else if ( IsAppWindow(frontmost) ) {
           te = ((DocumentPeek) frontmost)->te;
           if ( (*te)->selStart < (*te)->selEnd )
                  cutCopyClear = true;

           /* Cut, Copy, and Clear are enabled for application windows */
           /*     with selections                                      */

           paste = true;/* Paste is enabled for application windows    */
   }
   if ( undo )
      EnableItem(menu, iUndo);
   else
      DisableItem(menu, iUndo);
   if ( cutCopyClear ) {
      EnableItem(menu, iCut);
      EnableItem(menu, iCopy);
      EnableItem(menu, iClear);
   } else {
      DisableItem(menu, iCut);
      DisableItem(menu, iCopy);
      DisableItem(menu, iClear);
```

```
        }
    if ( paste )
       EnableItem(menu, iPaste);
    else
       DisableItem(menu, iPaste);
} /*AdjustMenus*/


/*  This is called when an item is chosen from the menu bar (after calling      */
/*      MenuSelect or MenuKey). It does the right thing for each command.       */

#define __SEG__ Main
void DoMenuCommand(menuResult)
    long        menuResult;
{
    short       menuID;
    short       menuItem;
    short       itemHit;
    Str255      daName;
    short       daRefNum;
    OSErr       error;
    OSErr       ignoreResult;
    TEHandle    te;

    menuID = HiWord(menuResult);         /* use macros for efficiency to get ..  */
    menuItem = LoWord(menuResult);       /*    menu item number and menu number  */
    switch ( menuID ) {
       case mApple:
              switch ( menuItem ) {
                     case iAbout:        /* bring up alert box for About         */

                            itemHit = Alert(rAboutAlert, nil);
                            break;
                     default:            /* all non-About items in this menu     */
                                         /*     are DAs                          */
                            GETITEM(GetMHandle(mApple), menuItem, &daName);
                            daRefNum = OPENDESKACC(&daName);
                            break;
              }
              break;
       case mFile:
              switch ( menuItem ) {
                     case iNew:
                            DoNew();
                            break;
                     case iClose:
                            DoCloseWindow(FrontWindow());
                            break;
                     case iQuit:
                            Terminate();
                                   break;
              }
              break;
       case mEdit:           /* call SystemEdit for DA editing and MultiFinder   */
```

```
                if ( ! SystemEdit(menuItem-1) ) {
                        te = ((DocumentPeek) FrontWindow())->te;
                        switch ( menuItem ) {
                                case iCut:
                                        TECut(te);              /* after cutting, export   */
                                                                /*    the TE scrap         */
                                        error = ZeroScrap();
                                        if ( error == noErr )
                                                ignoreResult = TEToScrap();
                                        break;
                                case iCopy:
                                        TECopy(te);             /* after copying, export   */
                                                                /*    the TE scrap         */
                                        error = ZeroScrap();
                                        if ( error == noErr )
                                                ignoreResult = TEToScrap();
                                        break;
                                case iPaste:                    /* import the TE scrap     */
                                                                /*    before  pasting      */
                                        error = TEFromScrap();
                                        if ( error == noErr )
                                                TEPaste(te);
                                        break;
                                case iClear:
                                        TEDelete(te);
                                        break;
                        }
                }
                break;
    }
    HiliteMenu(0);          /* unhighlight what MenuSelect or MenuKey highlighted */
}   /*DoMenuCommand*/


/* Create a new document and window. */

#define __SEG__ Main
void DoNew()
{
    Boolean   good;
    Ptr       storage;
    WindowPtr window;

    storage = NewPtr(sizeof(DocumentRecord));
    if ( storage != nil ) {
       window = GetNewWindow(rDocWindow, storage, (WindowPtr) -1);
       if ( window != nil ) {
               gNumDocuments += 1;             /* this will be decremented when  */
                                               /*    you call DoCloseWindow       */
               good = false;
               SetPort(window);
                       ((DocumentPeek) window)->te = TENew(&window->portRect,
                                                         &window->portRect);
```

```
            if ( ((DocumentPeek) window)->te != nil )
                good = true;                /* if TENew succeeded, the        */
                                            /*    document is good            */
            if ( good )
                ShowWindow(window);         /* if the document is good, make  */
                                            /*    the window visible          */
            else
                DoCloseWindow(window);      /* otherwise regret you ever      */
                                            /*    created it                  */
        } else
            DisposPtr(storage);             /* get rid of the storage if it   */
                                            /*    is never used               */
    }
}   /*DoNew*/


#define __SEG__ Main
void DoCloseWindow(window)
    WindowPtr window;

/* Close a window. This handles desk accessory and application windows. */

{
    TEHandle te;
    if ( IsDAWindow(window) )
       CloseDeskAcc(((WindowPeek) window)->windowKind);
    else if ( IsAppWindow(window) ) {
       te = ((DocumentPeek) window)->te;
       if ( te != nil )
           TEDispose(te);                        /* dispose the TEHandle    */
       DisposeWindow(window);
       gNumDocuments -= 1;
    }
} /*DoCloseWindow*/


/* Close the window that is passed and all windows behind it.  This is used to  */
/* close all the windows when the program quits, so it is in the Terminate      */
/* segment.  Note that it closes windows from back to front, by calling itself  */
/* recursively, which minimizes window updating.                                */

#define __SEG__ Terminate
void DoCloseBehind(window)
    WindowPtr window;
{
    if ( window != nil ) {              /* if you are passed a window, close    */
                                        /*    other windows behind it first     */
       DoCloseBehind((WindowPtr) (((WindowPeek) window)->nextWindow));
       DoCloseWindow(window);           /* now that all the windows behind are  */
                                        /*    closed, close this one            */
    }
}   /*DoCloseBehind*/
```

```
/* Clean up the application and exit. Close all of the windows so they can      */
/*    update their documents.                                                    */

#define __SEG__ Terminate
void Terminate()
{
    DoCloseBehind(FrontWindow());       /* close all windows */
    ExitToShell();
}   /*Terminate*/


/* Set up the whole world, including global variables, Toolbox managers, menus, */
/*   and a single blank document.                                               */

#define __SEG__ Initialize
void Initialize()
{
    OSErr     ignoreError;

    /* Ignore the error returned from SysEnvirons; even if an error occured, the */
    /* SysEnvirons glue will fill in the SysEnvRec.                              */

    ignoreError = SysEnvirons(sysEnvironsVersion, &gMac);
    if ( gMac.machineType < 0 )             /* old machines have...            */
       gHasWaitNextEvent = false;           /* no separate trap table; no      */
                                            /* WaitNextEvent                   */
    else
       gHasWaitNextEvent = TrapAvailable(_WaitNextEvent, ToolTrap);
    gInBackground = false;

    InitGraf((Ptr) &qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(nil);
    InitCursor();

    SetMenuBar(GetNewMBar(rMenuBar));       /* read menus into menu bar        */
    AddResMenu(GetMHandle(mApple), 'DRVR'); /* add DA names to Apple menu      */
    DrawMenuBar();

    gNumDocuments = 0;

    /* do other initialization here */

    DoNew();                                /* create a single empty document */
}   /*Initialize*/


/* Check if a window belongs to the application. */

#define __SEG__ Main
Boolean IsAppWindow(window)
```

```
    WindowPtr window;
{
    if ( window == nil )
       return  false;
    else                  /* application windows have non-negative windowKinds  */
       return  ((WindowPeek)  window)->windowKind  >=  0;
}   /*IsAppWindow*/


/* Check if a window belongs to a desk accessory. */

#define __SEG__ Main
Boolean IsDAWindow(window)
    WindowPtr window;
{
    if ( window == nil )
       return  false;
    else                                 /* DA windows have negative windowKinds */
       return  ((WindowPeek)  window)->windowKind  <  0;
}   /*IsDAWindow*/


/* Check to see if a given trap is implemented. This code is only used by the   */
/*  Initialize routine in this program, so it is in the Initialize segment.      */

#define __SEG__ Initialize
Boolean TrapAvailable(tNumber,tType)
    short     tNumber;
    TrapType  tType;
{
/* Check and see if the trap exists. On 64K ROM machines, tType will be ignored.*/

    return NGetTrapAddress(tNumber, tType)  != GetTrapAddress(_Unimplemented);
}   /* TrapAvailable */
```

Appendix B

A Pascal Example of a MultiFinder-Aware Application

The following Pascal program is an example of a MultiFinder-aware application.

```
{------------------------------------------------------------------------------}
{                                                                              }
{      MultiFinder-Aware Sample Application                                    }
{                                                                              }
{      Copyright © 1988 Apple Computer, Inc.                                   }
{      All rights reserved.                                                    }
{                                                                              }
{      This sample application was written by Macintosh Developer Technical    }
{      Support.  It displays a single, fixed-size window in which the user can }
{      enter and edit text.                                                    }
{                                                                              }
{------------------------------------------------------------------------------}


PROGRAM Sample;


{Segmentation strategy:

 This program consists of three segments. Main contains most of the code,
 including the MPW libraries, and the main program. Initialize contains
 code that is only used once, during startup, and can be unloaded after the
 program starts. %A5Init is automatically created by the Linker to initialize
 globals for the MPW libraries and is unloaded right away.}


{SetPort strategy:

 Toolbox routines do not change the current port.  However, this program uses a
 strategy of calling SetPort whenever you want to draw or make calls that
 depend on the current port.  This makes you less vulnerable to bugs in other
 software that might alter the current port (such as the  bug (feature?) in many
 desk accessories that change the port on OpenDeskAcc).  This strategy also makes
 the routines from this program more self-contained, since they don't depend on
 the current port setting.}


{Clipboard strategy:

 This program does not maintain a private scrap. Whenever a cut, copy, or paste
 occurs, you import/export from the public scrap to TextEdit's scrap right away,
 using the TEToScrap and TEFromScrap routines. If you did use a private scrap,
 the import/export would be in the activate/deactivate event and suspend/resume
 event routines.}


USES
      MemTypes, QuickDraw, OSIntf, ToolIntf;

CONST
      {MPW 3.0 will include a Traps.p interface file that includes constants for
       trap numbers.  These constants are from that file.}

      _WaitNextEvent             = $A860;
      _Unimplemented             = $A89F;

      {MaxOpenDocuments is used to determine whether a new document can be opened
       or created.  You keep track of the number of open documents, and disable the
       menu items that create a new document when the maximum is reached. If the
       number of documents falls below the maximum, the items are enabled again.}

      maxOpenDocuments           = 1;

      {SysEnvironsVersion is passed to SysEnvirons to tell it which version of the
       SysEnvRec is understood.}

      sysEnvironsVersion         = 1;

      {OSEvent is the event number of the suspend/resume and mouse-moved events sent
       by MultiFinder. Once you determine that an event is an osEvent, look at the
       high byte of the message sent with the event to determine which kind of
       osEvent it is. To differentiate suspend and resume events, check the
       resumeMask bit.}

      osEvent              = app4Evt;    {event used by MultiFinder}
      suspendResumeMessage= 1;           {high byte of suspend/resume event message}
      resumeMask           = 1;          {bit of message field for resume vs.suspend}
      mouseMovedMessage    = $FA;        {high byte of mouse-moved event message}

      {The following constants are all resource IDs. They correspond to resources
       in Sample.r.  See Appendix C. }

      rMenuBar      = 128;               {application's menu bar}
      rAboutAlert   = 128;               {about alert}
```

```
        rDocWindow    = 128;              {application's window}

        {The following constants are used to identify menus and their items. The menu
         constants are menu IDs, and the individual item constants are item numbers
         within the menus.}

        mApple        = 128;              {Apple menu}
        iAbout        = 1;

        mFile         = 129;              {File menu}
        iNew          = 1;
        iClose        = 4;
        iQuit         = 12;

        mEdit         = 130;              {Edit menu}
        iUndo         = 1;
        iCut          = 3;
        iCopy         = 4;
        iPaste        = 5;
        iClear        = 6;


TYPE
        {A DocumentRecord contains the WindowRecord for one of the document windows,
         as well as the TEHandle for the text being edited.  Other document fields
         can be added to this record as needed. For a similar example, see how the
         Window Manager and Dialog Manager add fields after the grafPort.}

        DocumentRecord = RECORD
                            window : WindowRecord;
                            te     : TEHandle;
        END;

        DocumentPeek  =  ^DocumentRecord;


VAR
        {GMac is used to hold the result of a SysEnvirons call. This makes
         it convenient for any routine to check the environment.}

        gMac                 : SysEnvRec;        {set up by Initialize}

        {GHasWaitNextEvent is set at startup, and tells whether the WaitNextEvent
         trap is available. If it is false, GetNextEvent must be called. }

        gHasWaitNextEvent    : BOOLEAN;          {set up by Initialize}

        {GInBackground is maintained by the osEvent handling routines. Any part of
         the program can check it to find out if it is currently in the background.}

        gInBackground        : BOOLEAN;          {maintained by Initialize and DoEvent}

        {GNumDocuments is used to keep track of how many open documents there are
         at any time. It is maintained by the routines that open and close documents.}
```

```
      gNumDocuments       : INTEGER;          {maintained by Initialize, DoNew, and
                                                      DoCloseWindow}


{$S Initialize}
FUNCTION TrapAvailable(tNumber: INTEGER; tType: TrapType): BOOLEAN;

{Check to see if a given trap is implemented. This is only used by the
 Initialize routine in this program, it is in the Initialize segment.}

BEGIN
{Check and see if the trap exists. On 64K ROM machines, tType will be ignored.}

      TrapAvailable := NGetTrapAddress(tNumber, tType) <>
                           GetTrapAddress(_Unimplemented);
END; {TrapAvailable}


{$S Main}
FUNCTION IsDAWindow(window: WindowPtr): BOOLEAN;

{Check if a window belongs to a desk accessory.}

BEGIN
      IF window = NIL THEN
            IsDAWindow := FALSE
      ELSE  {DA windows have negative windowKinds}
            IsDAWindow := WindowPeek(window)^.windowKind < 0;
END; {IsDAWindow}


{$S Main}
FUNCTION IsAppWindow(window: WindowPtr): BOOLEAN;

{Check if a window belongs to the application.}

BEGIN
      IF window = NIL THEN
            IsAppWindow := FALSE
      ELSE  {application windows have non-negative windowKinds}
            IsAppWindow := WindowPeek(window)^.windowKind >= 0;
END; {IsAppWindow}


{$S Main}
PROCEDURE DoCloseWindow(window: WindowPtr);

{Close a window. This handles desk accessory and application windows.}

BEGIN
      IF IsDAWindow(window) THEN
            CloseDeskAcc(WindowPeek(window)^.windowKind)
      ELSE IF IsAppWindow(window) THEN BEGIN
```

```
                WITH  DocumentPeek(window)^  DO
                        IF te <> NIL THEN
                                TEDispose(te);        {dispose the TEHandle}
                DisposeWindow(window);
                gNumDocuments := gNumDocuments - 1;
        END;
END; {DoCloseWindow}


{$S Main}
PROCEDURE DoNew;

{Create a new document and window.}

VAR
        good         : BOOLEAN;
        storage      : Ptr;
        window       : WindowPtr;

BEGIN
        storage := NewPtr(SIZEOF(DocumentRecord));
        IF storage <> NIL THEN BEGIN
                window := GetNewWindow(rDocWindow, storage, WindowPtr(-1));
                IF window <> NIL THEN BEGIN
                        gNumDocuments := gNumDocuments + 1;{ this will be decremented
                                                             when you call DoCloseWindow }
                        good := FALSE;
                        SetPort(window);

                        WITH window^, DocumentPeek(window)^ DO BEGIN
                                te := TENew(portRect, portRect);
                                IF te <> NIL THEN
                                        good := TRUE;      {if TENew succeeded, the
                                                              document is good }
                        END;
                        IF good THEN
                                ShowWindow(window)         {if the document is good, make
                                                              the window visible       }
                        ELSE
                                DoCloseWindow(window);     {otherwise regret you ever
                                                              created it}
                END ELSE
                        DisposPtr(storage);                {get rid of the storage if it
                                                              is never used}
        END;
END; {DoNew}


{$S Initialize}
PROCEDURE Initialize;

{Set up the whole world, including global variables, Toolbox managers, menus, and a
  single blank document.}
```

```
VAR
      ignoreError   : OSErr;

BEGIN
      {Ignore the error returned from SysEnvirons; even if an error occurred,
       the SysEnvirons glue will fill in the SysEnvRec.}

      ignoreError := SysEnvirons(sysEnvironsVersion, gMac);
      IF gMac.machineType < 0 THEN            {old machines have...}
            gHasWaitNextEvent := FALSE        {no separate trap table; no
                                                 WaitNextEvent}
      ELSE
            gHasWaitNextEvent := TrapAvailable(_WaitNextEvent, ToolTrap);
      gInBackground := FALSE;

      InitGraf(@thePort);
      InitFonts;
      InitWindows;
      InitMenus;
      TEInit;
      InitDialogs(NIL);
      InitCursor;

      SetMenuBar(GetNewMBar(rMenuBar));       {read menus into menu bar}
      AddResMenu(GetMHandle(mApple), 'DRVR'); {add DA names to Apple menu}
      DrawMenuBar;

      gNumDocuments := 0;

      {do other initialization here}

      DoNew;                                  {create a single empty document}
END; {Initialize}


{$S Terminate}
PROCEDURE DoCloseBehind(window: WindowPtr);

{Close the window that is passed and all windows behind it.  This is used to close
 all the windows when the program quits, so it is in the Terminate segment. Note
 that it closes windows from back to front, by calling itself recursively, which
 minimizes window updating.}

BEGIN
      IF window <> NIL THEN BEGIN       {if you are passed a window, close other
                                           windows behind it first}
            DoCloseBehind(WindowPtr(WindowPeek(window)^.nextWindow));
            DoCloseWindow(window);      {now that all the windows behind are closed,
                                           close this one}
      END;
END; {DoCloseBehind}


{$S Terminate}
```

```
PROCEDURE Terminate;

{Clean up the application and exit. Close all the windows so they can update their
documents.}

BEGIN
       DoCloseBehind(FrontWindow);       {close all windows}
       ExitToShell;
END;  {Terminate}


{$S Main}
PROCEDURE AdjustMenus;

{Enable and disable menus based on the current state.  This is called just
 before MenuSelect or MenuKey, so it can set up everything for the Menu Manager.
 Since these are the times that the user can see the menus or choose a menu item,
 you only need to enable/disable items then.}

VAR
       frontmost            : WindowPtr;
       menu                 : MenuHandle;
       undo                 : BOOLEAN;
       cutCopyClear         : BOOLEAN;
       paste                : BOOLEAN;

BEGIN
       frontmost            := FrontWindow;
       menu                 := GetMHandle(mFile);

       IF gNumDocuments < maxOpenDocuments THEN
              EnableItem(menu, iNew);     { New is enabled when you can open more }
                                          { documents }
       ELSE
              DisableItem(menu, iNew);
       IF frontmost <> NIL THEN           { Close is enabled when there is a      }
                                          ( window to close }
              EnableItem(menu, iClose)
       ELSE
              DisableItem(menu, iClose);

       menu                 := GetMHandle(mEdit);
       undo                 := FALSE;
       cutCopyClear         := FALSE;
       paste                := FALSE;

       IF IsDAWindow(frontmost) THEN BEGIN
              undo          := TRUE;      {all editing is enabled for DA windows   }
              cutCopyClear  := TRUE;
              paste         := TRUE;
       END ELSE IF IsAppWindow(frontmost) THEN BEGIN
              WITH DocumentPeek(frontmost)^.te^^ DO
                     IF selStart < selEnd THEN
                            cutCopyClear := TRUE;
                            { Cut, Copy, and Clear are enabled for application    }
                            { windows with selections}

            paste := TRUE;            {Paste is enabled for application windows}
        END;
        IF undo THEN
            EnableItem(menu, iUndo)
        ELSE
            DisableItem(menu, iUndo);
        IF cutCopyClear THEN BEGIN
            EnableItem(menu, iCut);
            EnableItem(menu, iCopy);
            EnableItem(menu, iClear);
        END ELSE BEGIN
            DisableItem(menu, iCut);
            DisableItem(menu, iCopy);
            DisableItem(menu, iClear);
        END;
        IF paste THEN
            EnableItem(menu, iPaste)
        ELSE
            DisableItem(menu, iPaste);
END; {AdjustMenus}


{$S Main}
PROCEDURE DoMenuCommand(menuResult: LONGINT);

{This is called when an item is chosen from the menu bar (after calling
 MenuSelect or MenuKey). It does the right thing for each command.}

VAR
        menuID              : INTEGER;
        menuItem            : INTEGER;
        itemHit             : INTEGER;
        daName              : Str255;
        daRefNum            : INTEGER;
        error               : OSErr;
        ignoreResult        : OSErr;
        te                  : TEHandle;

BEGIN
        menuID := HiWrd(menuResult);     {use built-ins (for efficiency)...}
        menuItem := LoWrd(menuResult);   {to get menu item number and menu number}
        CASE menuID OF
            mApple:
                CASE menuItem OF
                    iAbout:      {bring up alert box for About}
                    itemHit := Alert(rAboutAlert, NIL);
                    OTHERWISE BEGIN            {all non-About items in this
                                                menu are DAs}
                        GetItem(GetMHandle(mApple), menuItem, daName);
                        daRefNum := OpenDeskAcc(daName);
                        END;
                END;
        mFile:
                CASE menuItem OF
                        iNew:
                        DoNew;
                        iClose:
                        DoCloseWindow(FrontWindow);
                        iQuit:
                        Terminate;
                END;
        mEdit:                  {call SystemEdit for DA editing and MultiFinder}
                IF NOT SystemEdit(menuItem-1) THEN BEGIN
                        te := DocumentPeek(FrontWindow)^.te;
                        CASE menuItem OF
                                iCut: BEGIN
                                        TECut(te);    {after cutting, export the TE
                                                            scrap}
                                        error := ZeroScrap;
                                        IF error = noErr THEN
                                                ignoreResult := TEToScrap;
                                END;
                                iCopy: BEGIN
                                        TECopy(te);   {after copying, export the TE
                                                            scrap}
                                        error := ZeroScrap;
                                        IF error = noErr THEN
                                        ignoreResult := TEToScrap;
                                END;
                                iPaste: BEGIN   {import the TE scrap before
pasting}
                                        error := TEFromScrap;
                                        IF error = noErr THEN
                                                TEPaste(te);
                                END;
                                iClear:
                                        TEDelete(te);
                        END;
                END;
        END;
        HiliteMenu(0);          {unhighlight what MenuSelect or MenuKey highlighted}
END; {DoMenuCommand}


{$S Main}
PROCEDURE DrawWindow(window: WindowPtr);

{Draw the contents of an application window.}

BEGIN
        SetPort(window);
        TEUpdate(window^.portRect, DocumentPeek(window)^.te);
END; {DrawWindow}
```

```
{$S Main}
FUNCTION GetSleep: LONGINT;

{Calculate a sleep value for WaitNextEvent. This takes into account the things
 that DoIdle does with idle time.}

VAR
      sleep         : LONGINT;
      frontmost     : WindowPtr;

BEGIN
      sleep := MAXLONGINT;                      {default value for sleep}
      IF NOT gInBackground THEN BEGIN           {if you are in front...}
            frontmost := FrontWindow;           {and the front window is yours...}

            IF IsAppWindow(frontmost) THEN BEGIN
                  WITH DocumentPeek(frontmost)^.te^^ DO
                        IF selStart = selEnd THEN {and the selection is     }
                                                  {    an insertion point... }
                              sleep := GetCaretTime;    {you need to make the
                                                         insertion point blink}
            END;
      END;
      GetSleep := sleep;
END; {GetSleep}


{$S Main}
PROCEDURE DoIdle;

{This is called whenever you get a null event or a mouse-moved event.  It takes care
   of necessary periodic actions. For this program, it calls TEIdle.}

VAR
      frontmost     : WindowPtr;

BEGIN
      frontmost := FrontWindow;
      IF IsAppWindow(frontmost) THEN
            TEIdle(DocumentPeek(frontmost)^.te);
END; {DoIdle}


{$S Main}
PROCEDURE DoKeyDown(event: EventRecord);

{This is called for any keyDown or autoKey events, except when the Command key is
 held down. It looks at the frontmost window to decide what to do with the key
 pressed.}

VAR
      frontmost     : WindowPtr;
      key           : CHAR;
```

```
BEGIN
	frontmost := FrontWindow;
	IF IsAppWindow(frontmost) THEN BEGIN
		key := CHR(BAnd(event.message, charCodeMask));
		TEKey(key, DocumentPeek(frontmost)^.te);
	END;
END; {DoKeyDown}


{$S Main}
PROCEDURE DoContentClick(window: WindowPtr; event: EventRecord);

{This is called when a mouseDown occurs in the content of a window.}

VAR
	mouse		: Point;
	shiftDown	: BOOLEAN;

BEGIN
	IF IsAppWindow(window) THEN BEGIN
		SetPort(window);
		mouse := event.where;		{get the click position}
		GlobalToLocal(mouse);		{convert to local coordinates}
		shiftDown := BAnd(event.modifiers, shiftKey) <> 0;
		{extend if Shift is down}

		TEClick(mouse, shiftDown, DocumentPeek(window)^.te);
	END;
END; {DoContentClick}


{$S Main}
PROCEDURE DoActivate(window: WindowPtr);

{This is called when a window is activated.}

BEGIN
	IF IsAppWindow(window) THEN
		TEActivate(DocumentPeek(window)^.te);
END; {DoActivate}


{$S Main}
PROCEDURE DoDeactivate(window: WindowPtr);

{This is called when a window is deactivated.}

BEGIN
	IF IsAppWindow(window) THEN
		TEDeactivate(DocumentPeek(window)^.te);
END; {DoDeactivate}
```

```
{$S Main}
PROCEDURE DoUpdate(window: WindowPtr);

{This is called when an update event is received for a window.  It calls DrawWindow
  to draw the contents of an application window.}

BEGIN
       IF IsAppWindow(window) THEN BEGIN
              BeginUpdate(window);                    {this sets up the visRgn}
              IF NOT EmptyRgn(window^.visRgn) THEN    {draw if updating needs to be
                                                         done}
                     DrawWindow(window);
              EndUpdate(window);
       END;
END; {DoUpdate}


{$S Main}
PROCEDURE AdjustCursor(mouse: Point; region: RgnHandle);

{Change the cursor's shape, depending on its position. This also calculates a region
 that includes the cursor for WaitNextEvent.}

VAR
       frontmost     : WindowPtr;
       arrowRgn      : RgnHandle;
       iBeamRgn      : RgnHandle;
       iBeamRect     : Rect;

BEGIN
       frontmost := FrontWindow;  { only adjust the cursor when you are in front}

       IF (NOT gInBackground) AND (NOT IsDAWindow(frontmost)) THEN BEGIN
              {calculate regions for different cursor shapes}
              arrowRgn := NewRgn;
              iBeamRgn := NewRgn;

              {start arrowRgn wide open}
              SetRectRgn(arrowRgn, -32768, -32768, 32767, 32767);

              {calculate iBeamRgn}
              IF IsAppWindow(frontmost) THEN BEGIN
                     iBeamRect := DocumentPeek(frontmost)^.te^^.viewRect;
                     SetPort(frontmost);      {make a global version of the viewRect}

                     WITH iBeamRect DO BEGIN
                            LocalToGlobal(topLeft);
                            LocalToGlobal(botRight);
                     END;
                     RectRgn(iBeamRgn, iBeamRect);
              END;

              {subtract other regions from arrowRgn}
              DiffRgn(arrowRgn, iBeamRgn, arrowRgn);
```

```
                {change the cursor and the region parameter}
                IF PtInRgn(mouse, iBeamRgn) THEN BEGIN
                        SetCursor(GetCursor(iBeamCursor)^^);
                        CopyRgn(iBeamRgn, region);
                END ELSE BEGIN
                        SetCursor(arrow);
                        CopyRgn(arrowRgn, region);
                END;

                {get rid of local regions}
                DisposeRgn(arrowRgn);
                DisposeRgn(iBeamRgn);
        END;
END;  {AdjustCursor}


{$S Main}
PROCEDURE DoEvent(event: EventRecord);

{Do the right thing for an event. Determine what kind of event it is, and call
 the appropriate routines.}

VAR
        part   : INTEGER;
        window : WindowPtr;
        key    : CHAR;

BEGIN
        CASE event.what OF
                nullEvent:
                        DoIdle;
                mouseDown: BEGIN
                        part := FindWindow(event.where, window);
                        CASE part OF
                                inMenuBar: BEGIN
                                        AdjustMenus;
                                        DoMenuCommand(MenuSelect(event.where));
                                END;
                                inSysWindow:
                                        SystemClick(event, window);
                                inContent:
                                        IF window <> FrontWindow THEN BEGIN
                                                SelectWindow(window);

                                                {DoEvent(event);}
                                                {use this line for "do first click"}

                                        END ELSE
                                                DoContentClick(window, event);
                                inDrag:
                                        DragWindow(window, event.where, screenBits.bounds);
                                inGoAway:
                                        IF TrackGoAway(window, event.where) THEN
                        DoCloseWindow(window);
            END;
        END;
        keyDown, autoKey: BEGIN
            key := CHR(BAnd(event.message, charCodeMask));
            IF BAnd(event.modifiers, cmdKey) <> 0 THEN BEGIN
            {Command key down}

                IF event.what = keyDown THEN BEGIN
                    AdjustMenus;        {enable/disable/check menu
                                           items properly}
                    DoMenuCommand(MenuKey(key));
                END;
            END ELSE
                DoKeyDown(event);
        END;
        activateEvt: BEGIN
            window := WindowPtr(event.message);
            IF BAnd(event.modifiers, activeFlag) <> 0 THEN
                DoActivate(window)
            ELSE
                DoDeactivate(window);
        END;
        updateEvt:
            DoUpdate(WindowPtr(event.message));
        osEvent:
            CASE BSR(event.message, 24) OF   {high byte of message}
                mouseMovedMessage:
                    DoIdle;      {mouse moved is also an idle event}
                suspendResumeMessage: BEGIN
                    window := FrontWindow;
                    IF BAnd(event.message, resumeMask) <> 0 THEN BEGIN
                            gInBackground := FALSE;
                            DoActivate(window);
                            { Have to treat suspend/resume    }
                            { as deactivate/activate as well  }
                        END ELSE BEGIN
                            gInBackground := TRUE;
                            DoDeactivate(window);
                        END;
                    END;
            END;
    END;
END; {DoEvent}


{$S Main}
PROCEDURE EventLoop;

{Get events forever, and handle them by calling DoEvent.  Also call AdjustCursor
  each time through the loop.}

VAR
    cursorRgn    : RgnHandle;
```

```
        ignoreResult  : BOOLEAN;
        event         : EventRecord;

BEGIN
        cursorRgn := NewRgn;
        REPEAT
                IF gHasWaitNextEvent THEN
                        ignoreResult := WaitNextEvent(everyEvent, event, GetSleep,
                                                       cursorRgn)
                ELSE BEGIN
                        SystemTask;
                        ignoreResult := GetNextEvent(everyEvent, event);
                END;
                AdjustCursor(event.where, cursorRgn);
                DoEvent(event);
        UNTIL FALSE; {loop forever}
END; {EventLoop}


PROCEDURE _DataInit; EXTERNAL;

{This routine is automatically generated by the MPW Linker. This external reference
  to it is made so that its segment, %A5Init, can be unloaded.}


{$S Main}
BEGIN
        UnloadSeg(@_DataInit);      {note that _DataInit must not be in Main!}
        MaxApplZone;                {expand the heap so code segments load at the top}

        Initialize;                 {initialize the program}
        UnloadSeg(@Initialize);     {note that Initialize must not be in Main!}

        EventLoop;                  {call the main event loop}

END.
```

Appendix C

Resource Descriptions for the Example MultiFinder-Aware Application

Here are the resource descriptions for the MPW Rez tool used in Appendixes A and B.

```
/*----------------------------------------------------------------------------*/
/*                                                                            */
/*     Resources for the MultiFinder-Aware Sample Application                 */
/*                                                                            */
/*     Copyright © 1988 Apple Computer, Inc.                                  */
/*     All rights reserved.                                                   */
/*                                                                            */
/*----------------------------------------------------------------------------*/


#include "Types.r"

/* these #defines correspond to values in the Pascal and C source code        */

#define      rMenuBar     128                    /* application's menu bar  */
#define      rAboutAlert  128                    /* about alert */
#define      rDocWindow   128                    /* application's window    */

#define      mApple       128                    /* Apple menu              */
#define      mFile        129                    /* File menu               */
#define      mEdit        130                    /* Edit menu               */

/* we use an MBAR resource to load all the menus conveniently */

resource 'MBAR' (rMenuBar, preload) {
     { mApple, mFile, mEdit };        /* three menus */
```

```
};

resource 'MENU' (mApple, preload) {
       mApple, textMenuProc,
       0b1111111111111111111111111111101,       /* disable dashed line, enable About
                                                   and DAs */
       enabled, apple,
       {
              "About Sample…",
                     noicon, nokey, nomark, plain;
              "-",
                     noicon, nokey, nomark, plain
       }
};

resource 'MENU' (mFile, preload) {
       mFile, textMenuProc,
       0b0000000000000000000100000000000,       /* enable Quit only, program enables
                                                   others */
       enabled, "File",
       {
              "New",
                     noicon, "N", nomark, plain;
              "Open",
                     noicon, "O", nomark, plain;
              "-",
                     noicon, nokey, nomark, plain;
              "Close",
                     noicon, "W", nomark, plain;
              "Save",
                     noicon, "S", nomark, plain;
              "Save As…",
                     noicon, nokey, nomark, plain;
              "Revert",
                     noicon, nokey, nomark, plain;
              "-",
                     noicon, nokey, nomark, plain;
              "Page Setup…",
                     noicon, nokey, nomark, plain;
              "Print…",
                     noicon, nokey, nomark, plain;
              "-",
                     noicon, nokey, nomark, plain;
              "Quit",
                     noicon, "Q", nomark, plain
       }
};

resource 'MENU' (mEdit, preload) {
       mEdit, textMenuProc,
       0b0000000000000000000000000000000,       /* disable everything, program does
                                                   the enabling */
       enabled, "Edit",
        {
```

```
				"Undo",
					noicon, "Z", nomark, plain;
				"-",
					noicon, nokey, nomark, plain;
				"Cut",
					noicon, "X", nomark, plain;
				"Copy",
					noicon, "C", nomark, plain;
				"Paste",
					noicon, "V", nomark, plain;
				"Clear",
					noicon, nokey, nomark, plain
		}
};

/* this ALRT and DITL are used as an About screen */
resource 'ALRT' (rAboutAlert) {
		{40, 20, 160, 292}, rAboutAlert, {
				OK, visible, silent;
				OK, visible, silent;
				OK, visible, silent;
				OK, visible, silent
		};
};

resource 'DITL' (rAboutAlert) {
		 {
				{88, 180, 108, 260},
				Button {
					enabled,	"OK"
				};
				{8, 8, 24, 214},
				StaticText {
					disabled,	"MultiFinder-Aware Application"
				};
				{32, 8, 48, 237},
				StaticText {
					disabled,	"Copyright © 1988 Apple Computer"
				};
				{56, 8, 72, 136},
				StaticText {
					disabled,	"Brought to you by:"
				};
				{80, 24, 112, 167},
				StaticText {
					disabled,	"Macintosh Developer Technical Support"
				}
		}
};

resource 'WIND' (rDocWindow) {
		{64, 60, 314, 460},
		noGrowDocProc, invisible, goAway, 0x0, "untitled"
};
```

```
/* put the latest SIZE template here to rez with MPW 2.0 */

type 'SIZE' {
            boolean             dontSaveScreen,
                                saveScreen;
            boolean             ignoreSuspendResumeEvents,
                                acceptSuspendResumeEvents;
            boolean             enableOptionSwitch,
                                disableOptionSwitch;
            boolean             cannotBackground,
                                canBackground;
            boolean             notMultiFinderAware,
                                multiFinderAware;
            boolean             notOnlyBackground,
                                onlyBackground;
            boolean             dontGetFrontClicks,
                                getFrontClicks;
            unsigned bitstring[9] = 0;
            unsigned longint;   /* preferred memory size in bytes */
            unsigned longint;   /* minimum memory size in bytes */
};      /* ignore the warning caused by redefining SIZE */

/* here is the quintessential MultiFinder friendliness device, the SIZE resource */
resource 'SIZE' (-1) {
      dontSaveScreen,
      acceptSuspendResumeEvents,
      enableOptionSwitch,
      canBackground,                  /* You can background, although not     */
                                      /*  currently; your sleep value  */
                                      /*  guarantees that you don't hog the   */
                                      /*  Macintosh while you are in the      */
                                      /*  background.                         */

      multiFinderAware,               /* This says that you do your own       */
                                      /* activate/deactivate.  MultiFinder    */
                                      /*  does not trick the application.     */

      notOnlyBackground,              /* This is definitely not a background- */
                                      /*  only application!                   */

      dontGetFrontClicks,             /* Change this is if you want "do first */
                                      /*  click" behavior as in the Finder.   */

      60 * 1024,                      /* This (preferred) size is bigger than */
                                      /*  the minimum size so you can  */
                                      /*  have more text and scraps.          */

      40 * 1024                       /* A heap dump was viewed while the     */
                                      /*  program was running; it was using   */
                                      /*  about 27K, so 13K was added for     */
                                      /*  stack, text, and scraps.            */

};
```

Appendix D

The Notification Manager

The **Notification Manager,** in System version 6.0 and later, provides the user with an asynchronous "notification" service. It is especially useful for background applications running under MultiFinder that need to communicate with the user—since windows can easily be obscured by other applications. However, the Notification Manager can be used by any application; it is not limited to those applications that take advantage of the new MultiFinder environment.

Each application, desk accessory, or device driver can queue any number of notifications. For this reason, you should try to avoid posting multiple notifications, since each one will be presented separately to the user ("you have mail," "you have mail," ...).

Information describing each notification request is contained in the Notification Manager queue; you supply a pointer to a queue element describing the type of notification you desire. The Notification Manager queue is a standard Macintosh queue, as described in the Operating System Utilities chapter of *Inside Macintosh,* Volume II.

The Notification Manager provides a one-way communication path from the application to the user. There is no path from the user to the application. If you require this secondary communication link, do not use the Notification Manager. If, however, the Notification Manager provides what you want, but not exactly how you would like—say you wanted the application's icon to exhibit some special effect—then you should use the Notification Manager because in the future, such features may be possible.

Each entry in the Notification Manager queue is a static and nonrelocatable record of type **NMRec** with the following structure:

```
TYPE NMRec = RECORD
                qLink:          QElemPtr;     {next queue entry}
                qType:          INTEGER;      {queue type -- ORD(nmType) = 8}
```

```
        nmFlags:        INTEGER;        {reserved}
        nmPrivate:      LONGINT;        {reserved}
        nmReserved:     INTEGER;        {reserved}
        nmMark:         INTEGER;        {item to mark in Apple menu}
        nmSIcon:        Handle;         {handle to small icon}
        nmSound:        Handle;         {handle to sound record}
        nmStr:          StringPtr;      {string to appear in alert box}
        nmResp:         ProcPtr;        {pointer to response routine}
        nmRefCon:       LONGINT;        {for application use}
    END;
```

If you want to use the Notification Manager, you must also use SysEnvirons to test the System version. If the System is too old, put up an alert message to tell the user that System 6.0 or later is needed to run your application, and then exit gracefully.

How a notification happens

When a notification is handled, one or more of the following occurs (in this order):

1. the mark is put next to the application (or desk accessory) in the Apple menu
2. the icon is added to the list of icons that rotate with the Apple symbol in the menu bar
3. the sound is played
4. the dialog box is presented, and the user dismisses it
5. the response procedure is called

At this point, the mark in the Apple menu and the icon rotating with the Apple symbol in the menu bar will remain until the notification request is removed from the queue. The sound and the dialog box are only presented once.

Creating a notification request

To create a notification request, you must set up an NMRec with all the information about the notification you want:

- nmMark contains 0 for no mark in the Apple menu, 1 to mark the current application, or the refNum of a desk accessory to mark that desk accessory. An application should pass 1, a desk accessory should pass is its own refNum, and a driver should pass 0.
- nmSIcon contains nil for no icon in the menu bar, or a handle to a small icon to rotate with the Apple symbol. (A small icon is a 16x16 bitmap, often stored in a SIGN resource.) This handle does not need to be locked, but must be nonpurgeable.

- nmSound contains nil for no sound, –1 to use the system beep sound, or a handle to a sound record to be played with SndPlay. This handle does not need to be locked, but it must be nonpurgeable.
- nmStr contains nil for no alert, or a pointer to the string to appear in the alert message.
- nmResp contains nil if you don't want to supply a response procedure, –1 to use a predefined procedure that removes the request immediately after it is completed, or a pointer to a procedure that takes one parameter, a pointer to your queue element.

For example, this is how it would be declared if it were named MyResponse:

```
PROCEDURE MyResponse (nmReqPtr: QElemPtr);
```

❖ *Note:* When this response procedure is called, A5 and low-memory globals are not set up for you. If you need to access your application's globals, you should save your application's A5 in the nmRefCon field as discussed below.

Response procedures should never draw or do "user interface" things. You should wait until the application or desk accessory is brought to the front before responding to the user. Some good ways to use the response procedure are to dequeue and deallocate your Notification Manager queue element or to set an application global (being careful about A5) so that the application knows when the user has been notified.

You should probably use an nmResp of –1 with audible and alert notifications to remove the notification as soon as the sound has played or the alert box has been dismissed. You shouldn't use an nmResp of –1 with an nmMark or an nmSIcon, because the mark or icon would be removed before the user would see it. Note that an nmResp of –1 does not deallocate the memory block containing the queue element, it only removes it from the notification queue.

The nmRefCon routine is available for your use. One convenient way to use it is to put the application's A5 in this field so that the response procedure can access application globals. This is useful since the value of A5 is not guaranteed when the application calls the response procedure (see Chapter 2 for more information on the A5 world).

Notification Manager routines

The Notification Manager is automatically initialized each time the system starts up. To add a notification request to the notification queue, call NMInstall. When your application no longer wants a notification to continue, it can remove the request by calling NMRemove. NMInstall and NMRemove do not move or purge memory, and can be called from completion routines or interrupt handlers, as well as from the main body of an application and the response procedure of a notification request.

NMInstall

NMInstall adds the notification request specified by nmReqPtr to the notification queue. Here are the interface, glue, and result codes for NMInstall:

■ `FUNCTION NMInstall (nmReqPtr: QElemPtr) : OSErr;`

```
INLINE $205F, $A05E, $3E80;
```

Trap macro	_NMInstall ($A05E)
On entry	A0: theNMRec (pointer)
On exit	D0: result code (word)

NMInstall returns one of the result codes listed below.

Result codes:	noErr	No error
	nmTypErr (–299)	qType field isn't ORD(nmType)

❖ *Note:* qType must be set to ORD(nmType).

NMRemove

NMRemove removes the notification identified by nmReqPtr from the notification queue. Here are the interface, glue, and result codes for NMRemove:

■ `FUNCTION NMRemove (nmReqPtr: QElemPtr) : OSErr;`

```
INLINE $205F, $A05F, $3E80;
```

Trap macro	_NMInstall ($A05F)
On entry	A0: theNMRec (pointer)
On exit	D0: result code (word)

NMRemove returns one of the result codes listed below.

Result codes:	noErr	No error
	qErr	Not in queue
	nmTypErr (–299)	qType field isn't ORD(nmType)

❖ *Note:* qType must be set to ORD(nmType).

Appendix E

A Summary of the MultiFinder Traps

This appendix contains a summary listing of the new MultiFinder traps.

Temporary memory allocation calls

Here are the new MultiFinder temporary memory allocation calls.

- `FUNCTION MFFreeMem : LONGINT`

  `INLINE $3F3C, $0018, $A88F`

  MFFreeMem returns the total amount of free memory available for temporary allocation, in bytes.

- `FUNCTION MFMaxMem(VAR grow:Size) : Size`

  `INLINE $3F3C, $0015, $A88F`

  MFMaxMem compacts the MultiFinder heap zone, purges all purgeable blocks, and returns the number of bytes of the largest contiguous free block for temporary allocation.

- `FUNCTION MFTempNewHandle(logicalSize:Size;VAR resultCode:OSErr):Handle`

  `INLINE $3F3C, $001D, $A88F`

  MFTempNewHandle attempts to allocate a new relocatable block of logicalSize bytes for temporary usage and return a handle to it. The new block will be unlocked and unpurgeable. If an error occurs, MFTempNewHandle will return nil.

Result codes:	noErr	No error
	memFullErr	Not enough room

■ `FUNCTION MFTopMem: Ptr`

`INLINE $3F3C, $0016, $A88F`

MFTopMem returns a pointer to the top of your application's memory partition.

❖ *Note:* Do not use this call to calculate the size of your application's memory partition.

■ `PROCEDURE MFTempDisposHandle(h:Handle; VAR resultCode:OSErr)`

`INLINE $3F3C, $0020, $A88F`

MFTempDisposHandle releases the memory occupied by the relocatable block whose handle is h.

Result codes:	noErr	No error
	memWZErr	Attempt to operate on a free block

■ `PROCEDURE MFTempHLock(h:Handle; VAR resultCode:OSErr)`

`INLINE $3F3C, $001E, $A88F`

MFTempHLock locks the specified relocatable block, preventing it from being moved within the MultiFinder heap zone.

Result codes:	noErr	No error
	nilHandleErr	Nil master pointer
	memWZErr	Attempt to operate on a free block

■ `PROCEDURE MFTempHUnlock(h:Handle; VAR resultCode:OSErr)`

`INLINE $3F3C, $001F, $A88F`

MFTempHUnlock unlocks the specified relocatable block, allowing it to move.

Result codes:	noErr	No error
	nilHandleErr	Nil master pointer
	memWZErr	Attempt to operate on a free block

WaitNextEvent

The interface for WaitNextEvent is:

```
Function WaitNextEvent (eventMask      :  INTEGER;
                        VAR theEvent   :  EventRecord;
                        sleep          :  LongInt;     {  Tick Units  }
                        mouseRgn       :  RgnHandle ) : BOOLEAN;
```

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word. Proof pages were created on the Apple LaserWriter® Plus. Final pages were created on the Varityper® VT600™. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier, a fixed-width font.