Mac OS 8 Only Application

This paper describes a Mac OS 8 Only Application.

1.0 Definition

A Mac OS 8 Only Application is a new application type that explicitly takes advantage of Mac OS 8 technologies. It only uses preferred Mac OS 8 technologies (APIs).

This paper describes two distinct types of Mac OS 8 Only applications:

1. User Interface Application
2. Server programs

If you are concerned with binary or source compatibility (i.e you are starting from a System 7.x source base), see the *Mac OS 8 Transitional Application* document and/or the *Mac OS 8 Compatible Application* document.

1.1 User Interface Application

The User Interface Application is a classification of application which has windows, menus, dialogs, controls, etc. It often is document-centric and supports multiple windows. Most mainstream productivity applications fall into this category.

Two major technologies, Events and `HIObjects`, which have a profound affect on User Interface applications have changed in Mac OS 8.

1.1.1 Events

Overview

The code of a User Interface application is a collection of routines. In a Mac OS 8 application, these routines can be reached by Apple Event and SOM dispatching in addition to the traditional bindings provided by a compiler and linker. In Mac OS 8, Apple Event dispatching is the principle “plumbing” for connecting “black boxes” of routines together. An interface file (for example, HIWindows.idl, HILists.idl, or HIDialogs.idl) often represents a “black box” or component.

In a Mac OS 8 application, Apple Events are used for virtually all communication between the operating system components and the application. Apple Events are used to communicate low-level information, such as key presses and mouse movement, subsuming the role that low-level events played in System 7. Apple Events also continue to be used for higher-level semantic events, such as Open Document, Print, and Quit.

If unhandled by the application, low-level events are typically handled by various Mac OS 8 components that increase the meaning of the event by adding, refining, or otherwise transforming the information. Each component then produces an event with higher semantic meaning, which is handled by another component.

The developer of a Mac OS 8 application is often concerned with the point in each “event transformation pipeline” at which the application wants to tap in, with Apple Event handlers, to receive information. Generally, the further down the pipeline, the more the operating system is doing for the application.

In the simplest terms, a Mac OS 8 Only Application:

- Creates one or more Apple Event Handler Tables using `AENewHandlerTable()`,
- Installs one or more Apple Event Handlers into each table using `AEInstallHandler()`,
- Associates the table with an Apple Event Dispatcher using `AEPushDispatcherHandlerTable()`,
- If debugging, sets breakpoints in the handlers installed in the tables as needed,
- and gives control “forever” to the Apple Event Dispatcher using `AEReceive()`.

Application Startup

A Mac OS 8 application does not have an event loop in the traditional System 7 sense. Rather, after installing an initial set of process-wide Apple Event handlers, the application makes a single call to the AEReceive function, which retains control until an Apple Event handler returns a special error code, errAEReceiveTerminate. Typically the application's Quit handler will return this error code. Inside AEReceive, the Apple Event manager uses Mac OS 8 microkernel services to yield processor time until an event is available for the application. A simple Mac OS 8 application's startup code and Quit handler might look like this:

```
main()
{
       AEHandlerTableRef processTable;
       InitFonts();
       InitCursor();
       AENewHandlerTable(&processTable, 0);
       AEInstallHandler(processTable,
                        kCoreEventClass,
                        kAEQuitApplication,
                       HandleQuit,
                        0);
       AEPushDispatcherHandlerTable(
                      AEGetDefaultDispatcher(),
                      processTable);

       CreateMenus();

       AEReceive(AEGetDefaultDispatcher(),
                       kAEReceiveForever);

       return 0;
}

OSStatus HandleQuit(const AppleEvent *event,
          AppleEvent *reply, void *refCon,
          AEHandlerTableRef table)
{
       return errAEReceiveTerminate;
}
```

- The application first initializes the Font Manager and the system cursor. Note that a Mac OS 8 application no longer needs to call any of the other system initialization routines (`InitGraf, InitWindows, InitMenus, TEInit,` or `InitDialogs`). The QuickDraw shared library now exports a `QDGlobals` data structure for use by the current process, and InitGraf is called automatically for you by the system to initialize that structure; you do not need to define your own `QDGlobals`. The other Toolbox initialization routines are only provided for backwards compatibility with System 7 applications.
- Next, the application creates an `AEHandlerTable`, a new data type supported by the Mac OS 8 Apple Event Manager. System 7 also supported handler tables, but only in a limited, implicit fashion; there was only one handler table for the entire process, which the System 7 API `AEInstallEventHandler` modified. The Mac OS 8 Apple Event Manager supports multiple handler tables, which may be stacked or removed in LIFO order on an Apple Event dispatcher. The Apple Event dispatcher is a new structure that serves as an event target; there is a default dispatcher created automatically for each process, and an application may create additional dispatchers. When processing an Apple Event, the Apple Event Manager scans the stack of handlers on the process dispatcher, looking for the first table that contains a matching handler; that handler is then invoked with the event. (See the document *Apple Events in Mac OS 8* for more information.)
- After creating a handler table, the application installs a Quit handler and pushes the handler table onto the process dispatcher. This ensures that any Quit events received by the process will be sent to the application's handler.
- Next, the application creates its menus. Menus are discussed in more detail below, in the section on `HIObjects`.
- Finally, the application calls `AEReceive`, specifying that the Apple Event Manager should look for events on the process dispatcher, and that it should continue looking for events forever until a handler returns `errAEReceiveTerminate`. It's also possible to tell `AEReceive` to return after receiving just the next event.

To construct its menus, windows, and other user interface elements, a Mac OS 8 application uses the `HIObjects` class library. `HIObjects` are discussed next.

1.1.2 HIObjects

The `HIObjects` class library is a new object-oriented human interface toolkit provided by Mac OS 8. It entirely replaces the System 7 Window, Control, Dialog, Menu, and List Managers; a Mac OS 8 application will not use any of those managers at all.

The `HIObjects` class library is implemented using SOMObjects for Mac OS 8, the Macintosh implementation of IBM's System Object Model. SOM allows Apple to present a consistent, object-oriented programming model for all user interface elements, making it easier for application developers to learn and use the programming interfaces. It also allows Apple to modify and enhance the `HIObjects` class library in the future without breaking existing clients, unlike the System 7 Toolbox APIs, in which Apple was severely limited in the changes that could be made without compromising backwards compatibility. SOM is also designed to be language-independent, so the `HIObjects` class library may be used from both C and C++.

The `HIObjects` class library is so named because most classes in it descend from a single root class, `HIObject`. The descendants of `HIObject` are `HIWindow` and `HIPanel`. `HIWindows` replace the System 7 Window manager; they serve as containers for `HIPanels`. `HIPanels` and their descendants provide the actual user interface elements: all of the standard System 7 controls, a new set of controls provided in Mac OS 8, static items such as text captions and pictures, menus, lists, edit fields, and so on.

It's important to note that `HIObjects` are not an application framework, and are not meant to replace Apple or third-party application frameworks. An application framework typically provides a default implementation of an event loop, document support, a z-ordered and clipped view system for subdividing window content, and many other higher-level features. The `HIObjects` class library provides none of these features; it is strictly a user-interface toolkit that may be used by applications or by frameworks to build higher-level components and structures.

A Mac OS 8 application written in C might create an HIWindow like this:

```
static void CreateSampleWindow(Environment* ev,
                    Rect* bounds,
                    ResID titleID,
                    const StringPtr text,
                    AEHandlerTableRef windowTable)
{
```

```
        HIWindow* newWindow;
        RefLabel label={kHelloCreator, kDocumentID};

        newWindow = HIWindowNew();
        _InitWindow(newWindow, ev,
              &label,
              bounds,
              kHIWindowNormalClass,
              kHIWindowDocumentVariant,
              kHIWindowStandardDocumentAttributes
              | kHIWindowQuitOnClose,
           (HIRootPanel*) kHIWindowDefaultRootPanel,
             (HIWindow*) kHIFirstWindowOfClass);
        MyPushWindowEventTable(newWindow, ev,
                               windowTable);
        SetHIObjectTextTitle(newWindow,
                               ev, titleID);
        AddCollectionItem(
                 _GetCollection(newWindow, ev),
                 'TEXT',
                 0,
                 StrLength(text) + 1,
                 text);
        _Show(newWindow, ev);
}

static void MyPushWindowEventTable(HIWindow*
window, Environment* ev, AEHandlerTableRef
sourceTable)
{
        AEDispatcherRef dispatcher;
        AEHandlerTableRef tempTable;

        dispatcher = _GetEventDispatcher(window,
                                         ev);
        AEShareHandlerTable(sourceTable,
                            (void*) window,
                            &tempTable);
        AEPushDispatcherHandlerTable(dispatcher,
                                     tempTable);
}
```

- Because this sample is written in C, the application uses the HIWindowNew() function to allocate memory for the window. An application written in C++ could simply say "`new HIWindow`".
- After allocating the window, the application calls the window's `InitWindow` method to initialize it. All `HIObjects` share a common two-step initialization; first, the object is allocated; second, the object is initialized. No `HIObject` may be used before it is initialized. In this sample, the application specifies that:
    - this window should go in the normal document layer of windows (`kHIWindowNormalClass`)
    - its appearance should be that of a document (`kHIWindowDocumentVariant`)
    - it has a standard set of attributes, including close, zoom, collapse, and grow boxes, and that it automatically sends a Quit Apple Event when closed (`kHIWindowStandardDocumentAttributes` | `kHIWindowQuitOnClose`)
    - it should receive a default root panel automatically to hold any HIPanels that may be placed into the window (`kHIWindowDefaultRootPane`)
    - it should be positioned at the front of the window z-order (`kHIFirstWindowOfClass`)
    -
- As described in the Events discussion above, each process has an Apple Event dispatcher associated with it. Each window also has an Apple Event dispatcher associated with it, and Mac OS 8 automatically sends a predefined set of events to a window as necessary. When a window needs to be updated, for example, its dispatcher will receive an update Apple Event. In this example, the application has created an Apple Event handler table that will be used by each of its windows. After creating the window, the application uses its own utility routine `MyPushWindowEventTable` to clone the handler table and then push the cloned reference onto the window's dispatcher.
- After installing the event handler table, the application uses the `SetHIObjectTextTitle` routine to set the title of the window. The source code for this routine is provided on the WWDC '96 CD in the HelloWorldC sample application
- Next, the application associates some window-specific data with the window using the window's collection. The Collection Manager, originally provided with QuickDraw GX, is used by the `HIObjects` class library to provide each HIObject with an extensible collection of application-specific data.
- Finally, since all `HIObjects` are invisible when initially created, the application shows the window.

To create some simple menus, A Mac OS 8 Application might use the following code:

```
static void CreateMenus(Environment* ev)
{
        HIMenu* rootMenu;
        HIMenu* scratchMenu;
        HIStateChangeCallbackRef scratchRef;

        // menu bar
        rootMenu = HIMenuNew();
        _InitMenu(rootMenu, ev, NULL);
        _AddItems(rootMenu, ev, kHIFirstItem, 2);

        // Apple menu
        scratchMenu = HIMenuNew();
        _InitMenu(scratchMenu, ev, NULL);
        _AddItems(scratchMenu, ev,
                  kHIFirstItem, 1);
        _AddStateChangeCallback(scratchMenu, ev,
        (HIStateChangeCallbackProcPtr) MenuChanged,
                                &scratchRef);
        SetHIObjectTextTitle(scratchMenu,
                                ev, kAppleTitleID);
        SetupMenuItem(scratchMenu, ev, 0,
                    kAboutItemID, 0, kAboutID);
        _SetItemChild(rootMenu, ev,
                       0, scratchMenu);

        // File menu
        scratchMenu = HIMenuNew();
        _InitMenu(scratchMenu, ev, NULL);
        _AddItems(scratchMenu, ev,
                    kHIFirstItem, 1);
        _AddStateChangeCallback(scratchMenu, ev,
        (HIStateChangeCallbackProcPtr) MenuChanged,
                                &scratchRef);
        SetHIObjectTextTitle(scratchMenu,
                               ev, kFileTitleID);
        SetupMenuItem(scratchMenu, ev, 0,
                    kQuitItemID, 'Q', kQuitID);
```

```
        _SetItemChild(rootMenu, ev, 1,
                        scratchMenu);

        _SetRootHIMenu(ev, rootMenu);
        _Show(rootMenu, ev);
        _Draw(rootMenu, ev, NULL, NULL);
}
```

- In the `HIObjects` class library, the System 7 menu bar has been replaced by the concept of a root menu which contains other menus. The root menu is special in that it is drawn in the menu bar area of the screen; however, in other respects it is exactly the same as any other menu. The application therefore first creates and initializes a root menu, and adds two empty items to it to hold the Apple and File menus.
- Next, the application creates the Apple menu. It adds one empty to the menu, to hold the About menu item, and then calls the `SetupMenuItem` utility routine to set the contents of that item. The application also attaches a state-changed callback to the menu. Each `HIObject` has associated with it a list of state-changed callback functions that are called under predefined circumstances when part of the object's state changes. A menu, for example, calls its state-changed callbacks when a selection is made from the menu. The application's state-changed callback is shown below.
- After creating the Apple menu, the application creates the File menu in the same way, and then sets its root menu as the root menu known to the `HIObjects` class library. Finally it shows and draws the root menu.

```
static void SetupMenuItem(HIMenu* menu,
                          Environment* ev,
                          HIItemIndex item,
                          ResID textID,
                          UInt16 accelerator,
                          OSType code)
{
        RefLabel label;

        label.creator = kHelloCreator;
        label.id = code;
```

```
        SetListItemTextImage(menu, ev, item,
                             textID);
        if (accelerator != 0)
        {
             _SetItemAccelerator(menu, ev,
                   item, kHIAcceleratorCommand,
                  accelerator);
        }
        _SetItemRefLabel(menu, ev, item, &label);
}
```

- The application uses the `SetupMenuItem` utility function to prepare its menu items. `SetupMenuItem` uses the `SetListItemTextImage` routine to load the text of the menu item from a TextObject resource and install it into the menu title. The source code for this routine is provided on the WWDC '96 CD in the HelloWorldC sample application. `SetupMenuItem` also sets the `RefLabel` of the menu item. A `RefLabel` is a structure used by Mac OS 8 that contains two `OSTypes`. It is used as an identifier for a particular part of the system. The `HIObjects` class library provides space for a `RefLabel` for each `HIObject` and for each item in a menu or list. In this case, the application is using the `RefLabel` on the menu item to uniquely identify which item was selected in its state-changed callback:

```
static void MenuChanged(Environment* ev,
             HIStateChangeCodeCreator creator,
            HIStateChangeCode whatHappened,
             HIObject* object)
{
        RefLabel label;

        if (creator != kHIObjectAppleCreator ||
        whatHappened != kHIStateChangeItemSelected)
            return;

        _GetItemRefLabel((HIMenu*) object, ev,
                _GetSelectedItem((HIMenu*) object,
                                 ev),
                &label);
```

```
        switch (label.id)
        {
           case kAboutID:
             ShowAbout(ev);
             break;

           case kQuitID:
             SendQuit(true);
             break;

           default:
             break;
        }
}
```

- A state-changed callback receives two `OSTypes` classifying what happened: a creator code indicating the owner of the code, and a specific constant for each kind of state change. All state-changed codes produced by the `HIObjects` class library use `kHIObjectAppleCreator` as the creator code. In its state-changed callback, the application determines which menu item was selected by looking at the item's `RefLabel`. It then dispatches to the correct handler for that item.

This overview only describes a few areas of the `HIObjects` class library. For a more complete discussion, see the document *Human Interface Toolbox*, particularly chapter 1, "*Introduction to the Mac OS 8 Toolbox*."

1.2 Utility Applications

Utility applications are a class of User Interface applications that are not document-centric, but do make use of the user interface and do interact with the user.

Mac OS 8 will enable these types of applications by allowing the application to control several aspects of the user interface:

1. **MenuBar** - Mac OS 8 Only utility applications determine if and when the MenuBar should be displayed. The system will not assume that the MenuBar must be available.
2. **Process Menu** - Mac OS 8 Only utility applications determine if and when their process name should appear in the Process Menu. Coupled with the MenuBar control (above), the utility application can be user-accessible and controllable during part of its lifetime, and “invisible” during others.
3. **User Input Focus** - Mac OS 8 Only utility applications will be able to acquire and release the User Input focus. This allows the utility application to receive user input events (e.g. keyboard and mouse events) even when it is not the foreground process.

1.3 Factored Applications

Factored applications are a class of User Interface applications that are structured in such a way as to maximize user interface response and perceived performance. In addition, factored applications will typically make better use of the system by maximizing the advantages of preemptive multi-tasking.

Factoring Your App in System 7.x

The idea of factoring your application into a user interface (“front half”) part and the response (“engine”) part is not new to Mac OS 8. Applications that are scriptable and recordable are already factored to a degree. Mac OS 8 allows you to take a System 7 factored application and factor it further into separate preemptive tasks.

Mac OS 8 provides preemptive tasking, synchronization services and a reentrant Apple Event Manager. With these services, you can further separate your application into functional categories. In this scenario, you could send all computations to the computational task, the file system work to the filesystem task, etc.

If more fine-grained concurrency is desired, you can even consider a task oriented division where the "back-end" work associated with the user command is accomplished by a task. In this scenario, the task could be created dynamically to act upon the user’s command. When the work associated with this command is done, the task could terminate. Another alternative is to have a "pool" of tasks waiting to act on the next user command.

1.4 How To Build a Mac OS 8 Only User Interface Application

A Mac OS 8 Only application requires the interfaces and libraries on the **Mac OS 8 Developers Release**: ***Compatibility Edition*** CD. This release will provide helpful feedback while you are compiling, linking and running your application.

Interfaces

The Mac OS 8 version of the interfaces, like all Apple interfaces, are universal to all Apple software. These are the interfaces that Apple engineers use to write their software. The **Mac OS 8 Developers Release**: ***Compatibility Edition*** CD will include the latest version of our interfaces.

Libraries

In addition to the interfaces on the CD, we will include stub libraries on the CD to link your application against. These libraries correspond to the different types of products you might build. They allow you to link against one library without having to know what specific library the service (and symbol) in question came from.

Compiling Your Application

To link a Mac OS 8 Only application, use the **`BUILDING_FOR_SYSTEM8`** compiler flag to indicate to the system that you are building an application which only runs on Mac OS 8.

Linking Your Application

To link a Mac OS 8 Only application, use the **`AppMacOS8.stubs`** library in your development environment. Linking against this library ensures that you will not import facilities which are deprecated. The Mac OS 8 Only application does not link against InterfaceLib. It only links against libraries that are needed, and only requires these libraries and their dependent libraries at runtime. This allows Mac OS 8 Only applications which uses fewer system services to consume fewer system resources.

Running Your Application

When running your application against the debug version of the system release (on the **Mac OS 8 Developers Release: *Compatibility Edition***), you may encounter debugger breaks which detect unsupported or discouraged usage patterns. This will help you determine how well your application will run.

2.1 Server Programs

Mac OS 8 provides the foundation for a new Application Model, the Server. Servers are defined to be preemptive, faceless applications running in their own address space which are independent of the Macintosh Toolbox environment.

What should be written as a Server? Applications that:

- Are very performance sensitive
- Require a fully preemptive run-time environment
- Require the full protection of their own address space
- May need to run independent of an Application invocation
- May need to remain running across a Toolbox Restart
- Typically service multiple clients (applications &/or machines)
- Typically have no (or little) user interface

Examples of typical Server modeled Applications include: databases, mail handling, backup systems, ray-trace engines, personal information management systems, font Servers, etc. The closest thing (in principle) to Mac OS 8 Servers in System 7.x are faceless background applications. In addition, some existing System 7.x Extensions (INITs), Drivers (DRVRs) and Time Manager tasks are perfect candidates for re-writing as Mac OS 8 Servers.

Servers can be designed using a variety of Mac OS 8 System Services. The choice of which combination of services to use depends on what the Server is trying to do and what environment it requires.

2.1.1 Tasking Models

One of the first design choices one must make when writing a Server is whether the Server program uses one or multiple tasks.

Choosing how tasks are used to implement the Server determines the load handling characteristics and efficiency of that service. There are several basic options with many hybrids possible. The tasking options chosen will determine the requirements for synchronization (locking), request completion notification (async, sync), request dispatch, and protocol.

Single, Synchronous Task

The simplest task structure is a single task that handles one request at a time. No locking is needed and all requests are made synchronously. The

request dispatch is usually a FIFO and the client/Server protocol cannot support any circular dependencies. In this case, the Server cannot make a request that will result in another request to itself (since the Server makes all requests synchronously, it would end up waiting for a request to itself to complete – a request that it would never receive).

The main benefits of this tasking structure are simplicity and low system utilization. The downside is a lack of scalability in request handling. Since only one request is handled at a time, no parallelism is possible. However, if this is an occasionally used or stand-alone service with minimal performance requirements, then this task structure is a good choice.

This simple tasking model is probably a good choice for prototyping a Server program. Some services have no need of more than one task.

Single, Asynchronous Task

A variation on a single task structure uses asynchronous requests to implement parallelism within one task. A new request can be processed while waiting asynchronously for previous request processing to complete. This technique will only achieve parallelism if the service implements or uses IO bound processing (or other non-compute bound processing). This single task, asynchronous programming model generally uses less in terms of system resources than a multi-tasking model.

Some algorithms are simpler to implement in an asynchronous manner, others are more complex. For example, a file copy loop can be written very efficiently using asynchronous IO requests, whereas a multi-tasked version using synchronous IO requests would require a locking structure to manage access to shared buffers. On the other hand, if multiple layers of software are used to implement the service and the layers each independently make IO requests, then synchronous programming is much simpler because it allows straightforward exposition of the algorithm in each layer without the need for complex state machines.

Multi-Task Servers

There are many reasons why one might want to use multiple tasks in a Server (a.k.a. “threading” a Server). Two common reasons are complexity and performance.

Functional Multi-Tasking

Functional tasking uses a task for each functional component of a Server. For example, if a Server uses both a network connection and the local file system, one task would handle the network and another would perform the

file IO. When the processing requirements are non-uniform for all the types of requests a Functional Multi-Tasking model can work quite well.

Using multiple tasks for different areas of functionality can reduce complexity when the state associated with a functional component is solely managed by the task that services that component.

Load Balance Multi-Tasking

With Load Balance Multi-Tasking, every task can handle any request. Requests are distributed among a pool of tasks in order to balance load across the available system resources.

There are several possible algorithms for distributing requests among the various tasks. For example, if the protocol is connection oriented, then a task can be allocated for each connection. This preserves request order per connection and can greatly simplify synchronization and error recovery mechanisms.

The primary reason for using multiple-tasks in this manner is to maximize execution overlap. Generally, this will result in better performance.

2.1.2 Transport

Another design choice for Server writers is which transport to use. Mac OS 8 supports three different types of transports for Server programs: Open Transport, Apple Events and Microkernel Messaging.

Open Transport

Open Transport is Apple solution to transport-independent network. Open Transport based servers do not need to know what protocol they are using, and can provide a service over multiple network protocols (e.g. TCP/IP, AppleTalk, XNS,.etc.). Most network based services, such as FileShare servers and Web servers, should use Open Transport.

Open Transport based servers may also use Apple Events or Microkernel Messaging to receive local (non-network) requests.

For more information on Open Transport, see *Inside Macintosh: Open Transport.*

Apple Events

Apple Events allow a Server to be both transport independent and network independent. Servers based on Apple Events do not need to know that their clients are local or remote.

In addition, Apple Events provides a robust and high-performing data-model which deals with complex data types which do not lend themselves to Microkernel Messaging.

Apple Events are much more central to the programming model in Mac OS 8. See “Events” on page 38.

Sample Code

The following code is typical of an Apple Event based Server program (with error recover code removed for clarity):

```
OSStatus initialize(ServerID serverID)
{
     OSStatus status;
     AEDispatcherRef myRef;
     KernelProcessID myProcess =
                         CurrentKernelProcessID();

     myRef = AEGetDefaultDispatcher();

     // turn my default AEDispatcherRef into a
     // global AEDispatcherID
     status = AEGetEventDispatcherID(myRef,
                                  &gAEDispatcherID);

     // Create gSelfAddress
     status = AECreateDesc(typeKernelProcessID,
                                      &myProcess,
                       sizeof(KernelProcessID),
                                &gSelfAddress);

     status = AECreateAppleEvent(
                          kSimpleAEServerClass,
                          kClientEventID,
                          &gSelfAddress,
                          kAutoGenerateReturnID,
                          0,
                          &gaevt);
     status = AECreateAppleEvent(
                          kSimpleAEServerClass,
                          kTimerID,
                          &gSelfAddress,
                          kAutoGenerateReturnID,
                          0,
```

```
                                &gTimerAppleEvent);

    status = AECreateNotifier(&gTimerAppleEvent,
                                &gTimerNotification);

    status = initializeTimer();

    // let Server Mgr know that the server is up
    // and ready to take requests
    //   (pass in AEDisptcherID to Server Mgr for
    //    client lookup via ServerLookup())
    status = ServerCreated(serverID,
                            gAEDispatcherID);
    return(noErr);

}
```

The initialize routine does the work necessary to get the server ready to receive requests from a client (e.g. an application). While the actual initialization will vary from Server to Server, all Apple Event based Servers must call `ServerCreated` passing the `serverID` that they were passed in their main entry point, and the `AEDispatcherID` that they are "listening" to.

Here's a short description of the code above:

- `AEGetEventDispatcherID` is called to turn the `AEDispatcherRef` (which is only valid in the context in which it is returned) into a system-unique address, a `AEDispatcherID`.
- Next, `AECreateDesc` is called to create an address descriptor for use in subsequent `AECreateAppleEvent` calls.
- The next two `AECreateAppleEvent` calls are done at initialization time as an optimization. Since these events are reused, the are allocated at initialization time and not disposed. The use the `gSelfAddress` to address the `AppleEvents`.
- The call to `AECreateNotifier` is used to turn a `KernelNotification` into an `AppleEvent`. Many microkernel, IO, and Filesystem asynchronous calls use `KernelNotification` to inform the caller of the completion of the call. The converts this notification into an `AppleEvent` based notification.
- The `initializeTimer` routine is an example of how to use the microkernel `SetTimer` call with AppleEvents (see description below).

- Finally, `ServerCreated` is called. The `serverID` that is passed into main must be passed back to the Server Mgr along with the `AEDispatcherID` that clients should use to communicate with the Server.

The following code shows the `initializeTimer` routine mentioned above:

```
OSStatus initializeTimer()
{
     OSStatus status;
     AbsoluteTime expirationTime;

     // set expiration time to 3 seconds in the
     // future
     expirationTime = AddDurationToAbsolute(
                         3 * kDurationSecond,
                         UpTime());

     status = SetTimer(&expirationTime,
                         &gTimerNotification,
                         kNilOptions,
                         NULL);
     return(status);
}
```

The above code initializes a time (`AbsoluteTime`) and calls the microkernel call SetTimer. This call notifies the caller according to the microkernel notification record it is passed when the time it is passed has expired. In this case, the notification record (`KernelNotification`) has already been initialized by the `AECreateNotifier` call.

The following code shows a simple "main" routine of an AppleEvent server:

```
OSStatus ServerMain (ServerID serverID)
{
     OSStatus status;
     AEHandlerTableRef handlertableref;

     DebugStr("\pSimpleAEServer main()...");

     // initialization code
```

```
        status = initialize(serverID);

        // install my handlers
        status = AEGetDispatcherTopHandlerTable(
                            AEGetDefaultDispatcher(),
                            &handlertableref);

        status = AEInstallHandler(handlertableref,
                            kSimpleAEServerClass,
                            kTimerID,
                            HandleTimerEvent,
                            NULL);

        status = AEInstallHandler(handlertableref,
                            kSimpleAEServerClass,
                            kClientEventID,
                            HandleMyEvent,
                            NULL);

        // main event loop
        status = AEReceive(AEGetDefaultDispatcher(),
                            kAEReceiveForever);

        return (status);
}
```

Microkernel Messaging

Microkernel Messaging.

2.1.3 Macintosh Toolbox Usage by Servers

Mac OS 8 Servers are defined to be preemptive, faceless, applications which run completely independently of the Macintosh Toolbox environment. They are developed using the Mac OS 8 Only environment. As such, there are some typical Toolbox usage scenarios which would not be implemented the same way when developing a Server.

Some typical Toolbox usages and their replacement recommendations follow.

Graphics	No direct access to the screen is permitted for Servers. Off-screen rendering using the QuickDraw GX and QuickDraw 3D graphics systems is supported and a rendered image may be passed back to a facefull client for drawing to the screen.
Dialogs, Alerts, Menus, Controls, Windows, etc.	The HI Toolbox cannot be used by Servers. All HI interaction will need to occur via an Application frontend communicated to via Apple Events or microkernel messaging. User interface for Servers should be minimized and eliminated where possible.
Events	The Event Mgr is not available to Servers. Only Apple Events may be received, and the Mac OS 8 Apple Event dispatchers are the preferred means to receive them.
Resources	The Resource Manager is not available to Servers. Resources can only be read or written implicitly through other Apple provided services (e.g. Preferences Mgr) that make use of resources.
Memory	The Memory Manager is not available to Servers. Memory should be allocated using Dynamic Storage Allocation Services and memory allocators.
Files	The System 7 Files API is not available to Servers. The File Manager API should be used instead.
Notification Manager	The Notification Manager is available to Servers. User-visible notification may also be done via a front-end application communicated to via Apple Events.
Translation Manager	The Translation Manager is available to Servers.
Component Manager	The Component Manager is not available to Servers. Plug-in mechanisms should be implemented using either SOM or CFM directly.

Networking

Servers should use the following APIs to access the network.

- Open Transport
- Apple Events

Preferences and Configuration Files

Servers should use the Preferences Manager to maintain all configuration information. UI for configuration should be provided via an administrative application.

System Logging Services

The System Logging Service provides a centralized logging service for errors and informational messages. The Server implementor should attempt to put enough information in the log to make diagnosis of problems easy but not so much information that figuring out what is relevant is difficult. To avoid annoying the user, the log should be used in preference to the notification mechanism. In addition, a log message should always be generated before sending a notification to the user.

Servers should not create their own status log files. Cleanup and rotation of lots of log files is difficult and complicates Server upgrade.

2.2 How To Build a Mac OS 8 Only Server Program

A Mac OS 8 Only Server requires the interfaces and libraries on the **Mac OS 8 Developers Release**: ***Compatibility Edition*** CD. This release will provide helpful feedback while you are compiling, linking and running your Server.

Interfaces

The Mac OS 8 version of the interfaces, like all Apple interfaces, are universal to all Apple software. These are the interfaces that Apple engineers use to write their software. The **Mac OS 8 Developers Release**: ***Compatibility Edition*** CD will include the latest version of our interfaces.

Libraries

In addition to the interfaces on the CD, we will include stub libraries on the CD to link your Server against. These libraries correspond to the different types of products you might build. They allow you to link against one

library without having to know what specific library the service (and symbol) in question came from.

Compiling Your Server

To link your Server, use the **BUILDING_PREEMPTIVE_CODE** compiler flag to indicate to the system that you are building an Server which only runs on Mac OS 8.

Linking Your Server

To link a Mac OS 8 Only server, use the **`Server.stubs`** library in your development environment. Linking against this library ensures that you will not import facilities which are deprecated. Server programs do not link against InterfaceLib. As with Mac OS 8 Only applications, Servers only link against libraries that are needed, and only requires these libraries and their dependent libraries at runtime. This allows Servers which uses fewer system services to consume fewer system resources.

Running Your Server

When running your Server against the debug version of the system release (on the **Mac OS 8 Developers Release: *Compatibility Edition***), you may encounter debugger breaks which detect unsupported or discouraged usage patterns. This will help you determine how well your Server will run.

To facilitate development, you can place your Server in the “Mac OS Folder:Server Libraries” folder and it will be automatically launched each time the system is restarted.