5

CARBON AND UNIVERSAL PROCEDURE POINTERS

Introduction

Previous demonstration programs have called a function (AEInstallEventHandler) which takes a **universal procedure pointer** as a parameter; however, an explanation of this term has been delayed until this chapter because the AEInstallEventHandler calls were only incidental to the main purpose of those demonstrations. The demonstration programs at Chapter 7 are the first in which calls to a system software function which takes a universal procedure pointer as a parameter are central to the demonstration. As a brief but necessary prelude for what is to come, therefore, this chapter addresses the role of universal procedure pointers in the general scheme of things, when and why they were introduced, and their relevance in the Carbon environment.

In a sense, universal procedure pointers are a piece of historical baggage dragged into Carbon by Mac OS 8/9. They was first introduced with the so-called Universal Headers which, in turn, were introduced with the Power Macintosh. They had to with the ability of the Power Macintosh to run applications that use the instruction set of the Motorola 680x0 microprocessor (used in 680x0-based Macintoshes) as well as applications that use the native instruction set of the Power Macintosh's PowerPC microprocessor.

In the Chapter 7 demonstration programs, the system software function which takes a universal procedure pointer is TrackControl. This function is called by your application when a mouse-down event occurs in a control, such as a scroll bar. Prior to the introduction of the Power Macintosh and the associated introduction of the Universal Headers, the TrackControl prototype looked like this:

```
Sint16  TrackControl(TrackControl(ControlHandle theControl,Point localPoint,
                     ProcPtr actionProc);
```

The actionProc a parameter is simply the address of an application-defined function, called a **callback procedure** (or, in C terminology, a **callback function**) that is called repeatedly while the mouse button remains down. In other words, the TrackControl function used to take a **procedure pointer** (or, in C terminology, a **function pointer**) in its actionProc parameter.

The Universal Headers, which, amongst other things, allow you to write Classic API source code capable of being compiled as either 680x0 code or native PowerPC code, changed the prototype for TrackControl to:

```
ControlPartCode  TrackControl(ControlHandle theControl,Point startPoint,
                              ControlActionUPP actionProc);
```

Notice that the third parameter is now of type ControlActionUPP. This means that the actionProc parameter now takes a **universal procedure pointer**. This prototype has been carried through to Carbon, hence the necessity to gain a basic understanding of universal procedure pointers.

The 68LC040 Emulator and the Mixed Mode Manager

The Emulator

This ability of the Mac OS 8/9 system software to execute applications that use the instruction set of the Motorola 680x0 microprocessor as well as applications that use the native instruction set of the PowerPC microprocessor is provided by an **emulator** (the 68LC040 Emulator). The emulator, which is essentially a 680x0 microprocessor implemented in software, provides an execution environment that is virtually identical to the execution environment found on 680x0-based Macintoshes.

One important aspect of the 68LC040 emulator is that it made it possible for parts of the system software to remain as 680x0 code while other parts were progressively re-written, primarily for reasons of speed, as native PowerPC code. In this regard, it is important to understand that some of Mac OS 8/9 still remains as 680x0 code. For example, in Mac OS 8.6, parts of the Control Manager (including, possibly, `TrackControl`) remains in 680x0 code. For the purposes of explanation, the following assumes that, in Mac OS 8/9, `TrackControl` still exists as 680x0 code.

The Mixed Mode Manager

In Mac OS 8/9, the emulator works together with a manager called the **Mixed Mode Manager**. The Mixed Mode Manager manages **mode switches** between code in different **instruction set architectures (ISAs)**.

Mode Switches

In Mac OS 8/9, mode switches are required when an application calls a system software function (or, indeed, any other code) that exists in a different ISA. For example:

- When a PowerPC application invokes a system software function that exists only as 680x0 code, a mode switch is required to move from the native environment to the emulator environment. Then, when that system software function completes, another mode switch is required to return from the emulator to the PowerPC environment to allow the PowerPC application to continue executing.

  This situation can occur in a Carbon application running on Mac OS 8/9 because, as previously stated, not all of Mac OS 8/9 has been re-written as native PowerPC code.

- When a 680x0 application running under the emulator calls a system software function that exists as native PowerPC code, a mode switch is required to move out of the emulator and into the native PowerPC environment. Then, when that system software function completes, another mode switch is required to return to the emulator and to allow the 680x0 application to continue executing.

  This situation cannot occur in a Carbon application running on Mac OS 8/9 because all Carbon applications must be compiled as native PowerPC code.

The Mixed Mode Manager operates transparently to most applications and other types of software, meaning that most **cross-mode calls** (calls to code in a different ISA from the caller's ISA) are detected automatically by the Mixed Mode Manager and handled without intervention by the calling software.

Intervention in Mode Switching

Sometimes, however, executable code needs to interact directly with the Mixed Mode Manager to ensure that a mode switch occurs at the correct time. When writing native PowerPC code, you only have to intervene in the mode-switching process when you execute code whose ISA might be different from the calling code. For example, when you pass the address of your application-defined action function (native PowerPC code) to `TrackControl` (680x0 code), the ISA of the code whose address you are passing is different from the ISA of the function you are passing it to. In such cases, you must ensure that the Mixed Mode Manager is called to make the necessary mode switch. You do this by explicitly signalling:

- The type of code you are passing.
- The code's calling conventions.

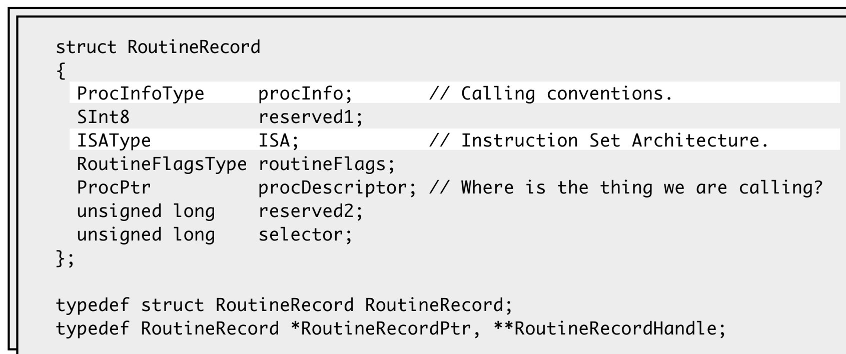
Indicating the ISA of a Callback Function — Routine Descriptors

You create a **routine descriptor** for a particular function to indicate the ISA of that function (see Fig 1). The first field of a routine descriptor (`goMixedModeTrap`) is an executable 680x0 instruction which invokes the Mixed Mode Manager. The Mixed Mode Manager having been called, it inspects the remaining fields of the routine descriptor to determine whether a mode switch is required. The Mixed Mode Manager is particularly interested in the `routineRecords` field.

The `routineRecords` field is an array of **routine structures**, each element of which describes a single function. In the simplest case, the array of routine structures contains a single element.

```
struct RoutineDescriptor
{
  unsigned short goMixedModeTrap;        // Mixed-mode A-Trap.
  SInt8          version;
  RDFlagsType    routineDescriptorFlags;
  unsigned long  reserved1;
  UInt8          reserved2;
  UInt8          selectorInfo;
  short          routineCount;
  RoutineRecord  routineRecords[1];       // The individual routines.
};

typedef struct RoutineDescriptor RoutineDescriptor;
```



```
struct RoutineRecord
{
  ProcInfoType     procInfo;       // Calling conventions.
  SInt8            reserved1;
  ISAType          ISA;            // Instruction Set Architecture.
  RoutineFlagsType routineFlags;
  ProcPtr          procDescriptor; // Where is the thing we are calling?
  unsigned long    reserved2;
  unsigned long    selector;
};

typedef struct RoutineRecord RoutineRecord;
typedef RoutineRecord *RoutineRecordPtr, **RoutineRecordHandle;
```

FIG 1 -THE ROUTINE DESCRIPTOR STRUCTURE AND A ROUTINE STRUCTURE

The `ISA` and `procInfo` fields are the most important fields in a routine structure:

- ***ISA Field.*** The `ISA` field contains the ISA code of the function being described, and always contains one of these two constants:

```
kM68kISA    = (ISAType) 0         MC680x0 architecture.
KPowerPCISA = (ISAType) 1         PowerPC Architecture.
```

  procInfo Field. The `procInfo` field contains the function's **function information**, including the function's calling conventions and information about the function's parameters.

Creating a Routine Descriptor For a Control Action Function

Using the function `NewControlActionUPP`, you can create a routine descriptor for a control action function as follows, in which `myControlAction` is your application-defined control action function:

```
ControlActionUPP myControlActionUPP;

myControlActionUPP = NewControlActionUPP(myControlAction);
```

Notice that the result returned by `NewControlActionUPP` is of type `ControlActionUPP`. The `UPP` stands for a **universal procedure pointer**, which is defined to be *either* a 680x0 function pointer *or* a pointer to a routine descriptor (hence the term "universal"). Thus, in Mac OS 8/9, the effect of the call to

`NewControlActionUPP` depends on whether it is executed in the 680x0 environment or the PowerPC environment:

- In the 680x0 environment, `NewControlActionUPP` simply returns its first parameter, that is, a pointer to your application-defined control action function.
- In the PowerPC environment, `NewControlActionUPP` creates a routine descriptor in your application heap and returns the address of that routine descriptor.

Effect of the Routine Descriptor

Once you have created the routine descriptor, you can later call `TrackControl` like this:

```
TrackControl(myControl,myPoint,myControlActionUPP);
```

In Mac OS 8/9, if your application is a PowerPC application (as will be the case in Carbon), the value passed in the `myControlActionUPP` parameter is not the address of your action function itself, but the address of the routine descriptor. If a 680x0 version of `TrackControl` executes your action function, it begins by executing the instruction in the first field of the routine descriptor. That instruction invokes the Mixed Mode Manager, which inspects the ISA of the action function (contained in the `ISA` field of the routine structure). Since that ISA differs from the ISA of the `TrackControl` function, the Mixed Mode Manager causes a mode switch. (Of course, if `TrackControl` existed as PowerPC code, the ISAs would be identical, and the Mixed Mode Manager would simply execute the action function without switching modes.)

In short, you solve the problem of indicating a routine's ISA by creating a routine descriptor and by using the address of that routine descriptor (that is, a universal procedure pointer) where you would have used the address of the function (that is, a procedure pointer) in the 680x0 programming environment.

Disposing of Routine Descriptors

Disposing of routine descriptors is only necessary or advisable if you know that you will not be using the descriptor any more during the execution of your application or if you allocate a routine descriptor for temporary use only.

Functions Requiring Routine Descriptors

Some of the typical functions for which you need to create routine descriptors are:

Function Type	*Examples are at the Demonstration Programs Associated With:*
Control action functions	Chapters 7, 14, 17, and 21.
Event filter functions	Chapters 4, 8, 14, and 21.
Apple event handling functions	Chapters 10, 18, and 26.
Key filter functions	Chapter 14.
Edit text validation functions	Chapter 14.
User pane drawing functions	Chapters 14 and 21.
User pane activate functions	Chapter 14.
Carbon event handlers	Chapters 17 to 26.
Carbon event timers	Chapter 17, 18, 21, 23, and 26.
Navigation Services event handlers	Chapter 18, 21, 23, and 26.
TextEdit click loop functions	Chapter 21.
List search functions	Chapter 22.
List definition functions	Chapter 22.
Drag and drop functions (various)	Chapter 23.
Device loop drawing functions	Chapter 25.

Universal Procedure Pointers and Carbon

Carbon supports universal procedure pointers transparently. By using the system-supplied universal procedure pointer functions, your application will operate correctly in both the Mac OS 8/9 and Mac OS X environments.

On Mac OS 8/9, the universal procedure pointer creation functions allocate routine descriptors in memory just as you would expect. On Mac OS X, the implementation of universal procedure pointers depends on various factors, including the object file format you choose. Universal procedure pointers will allocate memory if your application is compiled as a CFM binary, but are likely to return a simple procedure pointer if your application is compiled as a Mach-O binary. (All demonstration programs in this book are compiled as CFM binaries so that they will run on both Mac OS 8/9 and Mac OS X. Mach-O binaries only run on Mac OS X.)

In Carbon, routine descriptors must be disposed of using the specific disposal function associated with the creation function. For example, routine descriptors created with `NewControlActionUPP` must be disposed of using `DisposeControlActionUPP`. The generic disposal function `DisposeRoutineDescriptor` is not supported in Carbon.

Creation/Disposal Functions Relevant to Demonstration Programs

Creating Routine Descriptors

```
ControlActionUPP              NewControlActionUPP(ControlActionProcPtr userRoutine);
ModalFilterUPP                NewModalFilterUPP(ModalFilterProcPtr userRoutine);
AEEventHandlerUPP             NewAEEventHandlerUPP(AEEventHandlerProcPtr userRoutine);
ControlKeyFilterUPP           NewControlKeyFilterUPP(ControlKeyFilterProcPtr userRoutine);
ControlEditTextValidationUPP NewControlEditTextValidationUPP(ControlEditTextValidationProcPtr
                              userRoutine);
ControlUserPaneDrawUPP        NewControlUserPaneDrawUPP(ControlUserPaneDrawProcPtr userRoutine);
ControlUserPaneActivateUPP    NewControlUserPaneActivateUPP(ControlUserPaneActivateProcPtr
                              userRoutine);
TEClickLoopUPP                NewTEClickLoopUPP(TEClickLoopProcPtr userRoutine);
ListSearchUPP                 NewListSearchUPP(ListSearchProcPtr userRoutine);
ListDefUPP                    NewListDefUPP(ListDefProcPtruserRoutine);
DragTrackingHandlerUPP        NewDragTrackingHandlerUPP(DragTrackingHandlerProcPtr userRoutine);
DragReceiveHandlerUPP         NewDragReceiveHandlerUPP(DragReceiveHandlerProcPtr userRoutine);
DragSendDataUPP               NewDragSendDataUPP(DragSendDataProcPtr userRoutine);
DragInputUPP                  NewDragInputUPP(DragInputProcPtr userRoutine);
DragDrawingUPP                NewDragDrawingUPP(DragDrawingProcPtr userRoutine);
DeviceLoopDrawingUPP          NewDeviceLoopDrawingUPP(DeviceLoopDrawingProcPtr userRoutine);
```

Disposing of Routine Descriptors

```
void  DisposeControlActionUPP(ControlActionUPP userUPP);
void  DisposeModalFilterUPP(ModalFilterUPP userUPP);
void  DisposeAEEventHandlerUPP(AEEventHandlerUPP userUPP);
void  DisposeControlKeyFilterUPP(ControlKeyFilterUPP userUPP);
void  DisposeControlEditTextValidationUPP(ControlEditTextValidationUPP userUPP);
void  DisposeControlUserPaneDrawUPP(ControlUserPaneDrawUPP userUPP);
void  DisposeControlUserPaneActivateUPP(ControlUserPaneActivateUPP userUPP);
void  DisposeTEClickLoopUPP(TEClickLoopUPP userUPP);
void  DisposeListSearchUPP(ListSearchUPP userUPP);
void  DisposeListDefUPP(ListDefUPP userUPP);
void  DisposeDragTrackingHandlerUPP(DragTrackingHandlerUPP userUPP);
void  DisposeDragReceiveHandlerUPP(DragReceiveHandlerUPP userUPP);
void  DisposeDragSendDataUPP(DragSendDataUPP userUPP);
void  DisposeDragInputUPP(DragInputUPP userUPP);
void  DisposeDragDrawingUPP(DragDrawingUPP userUPP);
void  DisposeDeviceLoopDrawingUPP(DeviceLoopDrawingUPP userUPP);
```