23 Version 1.2 (Frozen)

PORTING TO THE POWER MACINTOSH

Introduction

The first system software release for Power Macintoshes provided the ability to execute applications that use the native instruction set of the Power Macintosh's PowerPC microprocessor *and* applications which use the 680x0 instruction set.

Although your 680x0 application can be run in emulation (see below) on a Power Macintosh, you will need to re-compile your application's source code into a PowerPC application, using a compiler capable of producing native PowerPC code, if you want to take advantage of the greater processing speed of the PowerPC microprocessor. Because the native PowerPC run-time environment is significantly different from that of the 680x0, you may need to re-write certain areas of your code before you can compile it as PowerPC code.

This chapter addresses matters relevant to running 680x0 applications on a Power Macintosh and to modifying the source code of an existing 680x0 application so that the application can be compiled into a PowerPC application.

The 68LC040 Emulator

The capability to execute applications which use the PowerPC instruction set *and* applications which use the 680x0 instruction set is provided by an **emulator** (the 68LC040 Emulator). The emulator provides an execution environment which is virtually identical to the execution environment found on 680x0-based Macintoshes. More specifically, the emulator:

- Converts 680x0 instructions into PowerPC instructions and issues those instructions to the PowerPC processor.
- Updates the emulated environment (such as the emulated 680x0 registers) in response to the operations of the PowerPC processor.

Under the emulator, all existing 680x0 applications and other software modules will execute without modification on Power Macintoshes provided that they:

- Are 32-bit clean.
- Are compatible with operations of the Virtual Memory Manager.
- Are able to operate smoothly in the cooperative multitasking environment maintained by the Process Manager.

- Conform to the general requirements of Macintosh software as documented in Inside Macintosh.

The emulator also makes it possible for parts of the system software to remain as 680x0 code while other parts of the system software are re-implemented, primarily for reasons of speed, as native PowerPC code. (The Memory Manager and QuickDraw, for example, were re-implemented as native PowerPC in the first system software release for Power Macintoshes.)

The emulator has some limitations. Possibly the most significant of these is that it does not support the instruction sets of the 68881 or 68882 co-processors or of the 68851 PMMU.

The Mixed Mode Manager

The emulator works together with a new manager called the **Mixed Mode Manager**. The Mixed Mode Manager manages **mode switches** between code in different **instruction set architectures (ISAs)**, switching the execution context between the CPU's native PowerPC context and the 68LC040 emulator.

Mode Switches

Mode switches are required when an application calls a system software routine (or, indeed, any other code) that exists in a different ISA. For example:

- When a 680x0 application running under the emulator calls a system software routine 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 routine completes, another mode switch is required to return to the emulator and to allow the 680x0 application to continue executing.

- When a PowerPC application invokes a system software routine or other code 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 routine completes, another mode switch is required to return from the emulator to the PowerPC environment to allow the PowerPC application to continue executing.

The Mixed Mode Manager is intended to operate 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.

Occasionally, however, some executable code needs to interact directly with the Mixed Mode Manager to ensure that a mode switch occurs at the correct time. Because the emulator is designed to allow existing 680x0 applications to execute without modification, it is always the responsibility of native applications to implement any changes necessary to interact with the Mixed Mode Manager.

Intervention in Mode Switching

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 a callback routine to the system software, it is possible that the ISA of the code whose address you are passing is different from the ISA of the routine 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.

An Example - Callback Routines

As an example, suppose you are writing a native PowerPC application that calls the Control Manager routine `TrackControl`. Recall from Chapter 5 — Controls that `TrackControl` accepts as one of its parameters the address of an action procedure that is called repeatedly while the mouse button remains down. `TrackControl` has no way of determining in advance the ISA of the code whose address you will pass to it. Moreover, your application has no way of determining in advance the ISA of the `TrackControl` routine, so you cannot know whether your action procedure and the `TrackControl` procedure are of the same ISA.

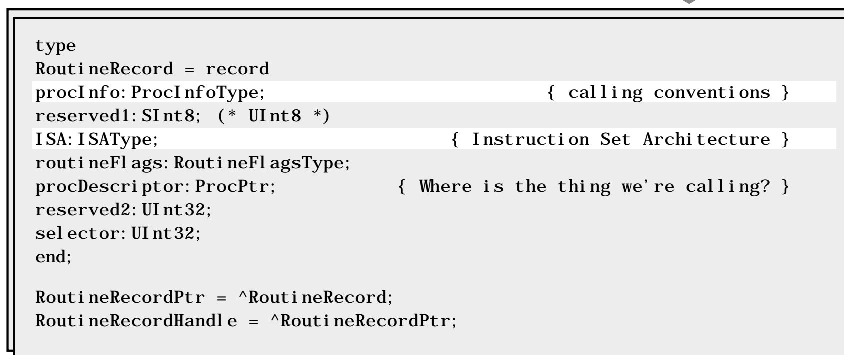
Because of all this, you must explicitly indicate the ISA of any callback routines whose addresses you pass to system software routines such as `TrackControl`.

Indicating the ISA of a Callback Routine — Routine Descriptors

You indicate the ISA of a particular routine by creating a **routine descriptor** for that routine (see Fig 1). The first field of a routine descriptor (`goMixedModeTrap`) is an executable 680x0 instruction which invokes the Mixed Mode Manager. When the Mixed Mode Manager is called, it inspects the remaining fields of the routine descriptor — in particular, the `routineRecords` field — to determine whether a mode switch is required. The `routineRecords` field is an array of **routine records**, each element of which describes a single routine. In the simplest case, the array of routine records contains a single element.

```
type
RoutineDescriptor = packed record
goMixedModeTrap: UInt16;                          { Mixed-mode A-Trap }
version: SInt8;
routineDescriptorFlags: RDFlagsType;
reserved1: UInt32;
reserved2: UInt8;
selectorInfo: UInt8;
routineCount: UInt16;
routineRecords: array [0..0] of RoutineRecord;{ The individual routines }
end;

RoutineDescriptorPtr = ^RoutineDescriptor;
RoutineDescriptorHandle = ^RoutineDescriptorPtr;
```



```
type
RoutineRecord = record
procInfo: ProcInfoType;                       { calling conventions }
reserved1: SInt8; (* UInt8 *)
ISA: ISAType;                          { Instruction Set Architecture }
routineFlags: RoutineFlagsType;
procDescriptor: ProcPtr;          { Where is the thing we're calling? }
reserved2: UInt32;
selector: UInt32;
end;

RoutineRecordPtr = ^RoutineRecord;
RoutineRecordHandle = ^RoutineRecordPtr;
```

FIG 1 -THE ROUTINE DESCRIPTOR RECORD AND ROUTINE RECORD

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

- `ISA` **Field.** The `ISA` field encodes the ISA of the routine being described. It always contains one of these two constants, which are defined in the Universal Interfaces file MixedMode.p:

```
kM68kISA   =  0    { MC680x0 architecture. }
kPowerPCISA =  1    { PowerPC Architecture. }
```

- `procInfo` **Field.** The `procInfo` field contains the routine's **procedure information**, which encodes the routine's calling conventions and information about the number and location of the

routine's parameters. For the standard kinds of callback routines and other types of "detached" code, the Universal Interfaces files include definitions of procedure information. For example, the interface file Controls.p includes this definition:

```
uppControlActionProcInfo = $000002C0; { procedure (4 byte param, 2 byte param); }
```

This procedure information specification indicates that a control action procedure follows standard Pascal calling conventions and takes two stack-based parameters (a control handle and a part code) and returns no result.

Creating a Routine Descriptor

You can create a routine descriptor by calling the Mixed Mode Manager function `NewRoutineDescriptor`, for example:

```
myActionProc : UniversalProcPtr;
myActionProc := NewRoutineDescriptor(ProcPtr(myAction),uppControlActionProcInfo,
                                     GetCurrentISA());
```

In this example, `myAction` is the address of your control action procedure and `GetCurrentISA` is a constant (determined at compile-time) which represents the current ISA. The effect of a call to `NewRoutineDescriptor` depends on whether it is executed in the 680x0 environment or the PowerPC environment:

- In the 680x0 environment, `NewRoutineDescriptor` simply returns its first parameter.
- In the PowerPC environment, `NewRoutineDescriptor` creates a routine descriptor in your application heap and returns the address of that routine descriptor.

Notice that the result returned by `NewRoutineDescriptor` is of type `UniversalProcPtr`. A **universal procedure pointer** is defined in the Universal Interfaces file Types.p to be a procedure pointer:

```
ProcPtr = Ptr;
Register68kProcPtr = ProcPtr;  { procedure ; }
ProcHandle = ^ProcPtr;
UniversalProcPtr = ProcPtr;
UniversalProcHandle = ^ProcPtr;
```

Effect of the Routine Descriptor

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

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

If your application is a PowerPC application, the value passed in the `myActionProc` parameter is not the address of your action procedure itself, but the address of the routine descriptor. When a 680x0 version of `TrackControl` executes your action procedure, 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 routine (contained in the `ISA` field of the routine record). If that ISA differs from the instruction set architecture of the `TrackControl` routine, the Mixed Mode Manager causes a mode switch. Otherwise, if the ISAs are identical, the Mixed Mode Manager simply executes the action procedure without switching modes.

Routines Requiring Routine Descriptors

Thus you satisfy the requirement to explicitly indicate a routine's ISA by creating routine descriptors and by using the address of those routine descriptors where you would have used procedure pointers in the 680x0 programming environment.

Remember, however, that you only need to do this when you need to pass the address of a routine to some *external* piece of code (such as a system software routine or some other application) that might be in a different ISA from that of the routine. You do not need to create routine descriptors for routines that are called only by your application. More generally, if you know for certain that a routine is

always called by code of the same ISA, you can and should continue to use procedure pointers instead of universal procedure pointers.

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

- Control action procedures.
- Event filter functions.
- VBL tasks.

Procedure Information Definitions, and Other Routines for Creating Routine Descriptors

The Universal Interfaces changed all references to parameters of type `ProcPtr` to references of type `UniversalProcPtr`. In addition, the Universal Interfaces:

- Contain procedure information definitions for all of the standard kinds of callback routines.
- Define new routines for creating routine descriptors which you can use in lieu of the more generalised method using `NewRoutineDescriptor` shown above. For example, the Universal Interfaces file `Controls.p` contains this definition for the `NewControlActionProc` function:

```
{$IFC PROCTYPE }
 ControlActionProcPtr = procedure (theControl: ControlRef; partCode:
ControlPartCode);
{$ELSEC}
 ControlActionProcPtr = ProcPtr;  { procedure ControlAction(theControl: ControlRef;
partCode: ControlPartCode); }
{$ENDC}

 ControlActionUPP = UniversalProcPtr;

function NewControlActionProc(userRoutine: ControlActionProcPtr):
                                                    ControlActionUPP;
```

This enables you to replace the previous example with this simpler way to create the routine descriptor for a control action procedure:

```
myActionUPP : ControlActionUPP;
myActionProc := NewControlActionProc(ControlActionProcPtr(@myAction));
```

Disposing of Routine Descriptors

Routine descriptors may be disposed of using `DisposeRoutineDescriptor`, although this 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.

The PowerPC Native Environment

In the emulation environment provided by the 68LC040 emulator, the organisation of an application partition, and the run-time behaviour of emulated software are identical to that provided on 680x0-based Macintoshes. However, the run-time environment[1] for native PowerPC software is significantly different from that of standard 680x0 run-time environment. The PowerPC environment provides a much simpler run-time model made possible by the use of **fragments** as the standard way of organising executable code and data in memory

[1]A run-time environment is a set of conventions which determine how code is to be loaded into memory, where it is to be stored, how it is to be addressed, and how functions call other functions and system software routines.

Fragments

A fragment is any block of executable PowerPC code and its associated data. Fragments can be of any size, and are complete, executable entities[2]. Amongst other things, they use a method of addressing the data they contain that is different and more general than the A5-related method used by 680x0 applications to address their global data (see Chapter 1 — System Software, Memory, and Resources). The natural consequence of this is that any PowerPC software packaged as a fragment has easy access to global data[3]. In the PowerPC environment, any routine contained in an application has automatic access to the application's global variables.

Categories of Fragments

There are three broad categories of fragments:

- **Applications.** An application is a fragment which can be launched by the user from the Finder.
- **Import Libraries.** An import library is a fragment which contains code and data associated with some other fragment or fragments. The system software, for example, is an import library. When you link an import library with your application, the import library's code is not copied to your application; rather, your application contains symbols known as imports which refer to some code or data in the import library.
- **Extensions.** An extension is a fragment which extends the capabilities of some other fragment. For example, your application might use external code modules like control definition functions. Unlike import libraries, extensions must be explicitly connected to your application during execution. There are two types of extensions, namely, **application extensions** and **system extensions**. An application extension is an extension that is used by a single application.

Import libraries and system extensions are sometimes called **shared libraries**, because the code and data they contain can be shared by multiple clients.

Fragment Storage and Loading

The physical storage for a fragment is a **container**, which can be any kind of object accessible by the operating system. The system software import library, for example, is stored in ROM. The fragment containing an application's executable code is stored in the application's data fork, which is a file of type 'APPL'. A container can also be a resource.

The process of loading a fragment into memory and preparing it for execution is handled by the Code Fragment Manager. The code and data sections of a loaded fragment are loaded into separate sections of memory, which are generally not contiguous. Regardless of where it is loaded, however, there is no segmentation within the code section of a fragment.

Even though a fragment's code and data sections can be loaded anywhere in memory, those sections cannot be moved in memory once they have been loaded[4].

Container Formats. The Code Fragment Loader recognises two kinds of container format:

- **Extended Common Object File Format (XCOFF).** XCOFF is a refinement of COFF, the standard executable file format of many UNIX-based computers. XCOFF is supported primarily because the early development tools produce executable code in the XCOFF format.

---

[2]The term "fragment" was chosen to avoid confusion with the terms already used in Inside Macintosh to describe executable code, such as "component" and "module". The term is not intended to suggest that the block of code and data is small, detached or incomplete.

[3]In the 680x0-based system software, it is sometimes difficult to use global data within types of software other than applications. In addition, it is often complicated for a routine installed by some code to gain access to the code's global variables. For example, you cannot, in the current 680x0 environment, write a VBL task that uses your application's global variables without somehow passing your application's A5 value to the VBL task (see Chapter 19 — Custom Control Definition Functions and VBL Tasks).

[4]In the 680x0 environment, an application's code can be unloaded (by the Memory Manager) and later re-loaded into a different place in memory.

- **Preferred Executable Format (PEF).** PEF is an object file format developed by Apple Computer. A container in the PEF format is dramatically smaller than the corresponding container in the XCOFF format. PEF provides support for a fragment's optional initialisation and termination routines and for the version checking performed by the Code Fragment Manager when an import library is connected to a fragment.

Code Fragment Resource

As previously stated, the first version of the system software for PowerPC-based Macintosh computers allows the user to run both 680x0 and PowerPC applications. Accordingly, the Process manager needs some method of determining, at the time the user launches the application, what kind of application it is. The Process Manager assumes that the application is a 680x0 application unless you indicate otherwise. You do this, by including in the resource fork of your PowerPC application, a **code fragment resource** (resource type `'cfrg'`) with a resource ID of 0. This resource indicates:

- The ISA of your application's executable code.
- The location of the code's container.

A typical code fragment resource, in Rez input format, is as follows:

```
#include "CodeFragment.Types.r"
resource 'cfrg' (0)
{
  {
    kPowerPC,          /* Instruction set architecture */
    kFullLib,          /* No update level for apps */
    kNoVersionNum,     /* No implementation version number */
    kDefaultStackSize  /* No definition version number */
    kNoAppSubFolder,   /* Use default stack size */
    kIsApp,            /* No library directory */
    kOnDiskFlat,       /* Fragment is an application */
    kZeroOffset,       /* Fragment is on disk */
    kWholeFork,        /* Fragment starts at fork start */
    "My Application"   /* Fragment occupies entire fork */
  }
};
```

Amongst other things, this resource specification indicates that the application consists of PowerPC code, the code is contained in the application's data fork, and the code container occupies the entire data fork.

You do not have to create the `'cfrg'` resource yourself because CodeWarrior does this for you.

Effect of the Code Fragment Resource

Typically, the code and data for a PowerPC application are contained in your application's data fork, as shown at Fig 2. If your application does not contain a code fragment resource, the Process Manager assumes that your application is a 680x0 application and calls the Segment Manager to load your application's executable code from resources of type `'CODE'` in your application's resource fork (see Fig 2).

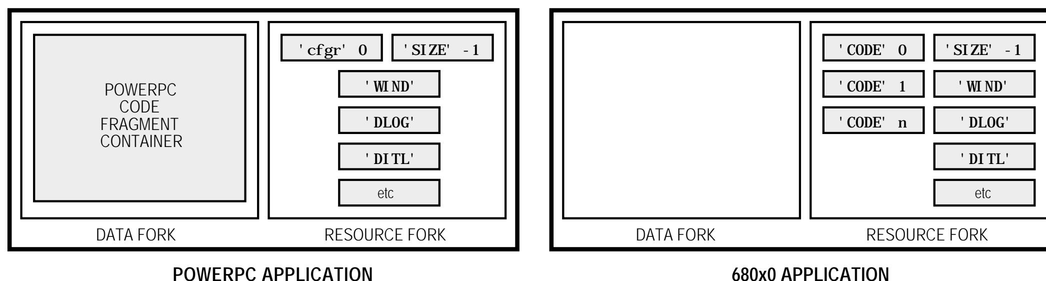


FIG 2 - THE STRUCTURE OF 680x0 AND POWERPC APPLICATIONS

Fat Applications

The placement of an application's PowerPC code in the data fork makes it easy to create a **fat application**. Fat applications contain both PowerPC code and 680x0 executable code, as shown at Fig 3.[5]

Chapter 2 (Application Projects) of the CodeWarrior manual Targeting Mac OS contains a section (Creating FAT Applications) outlining the steps required to create a fat application.

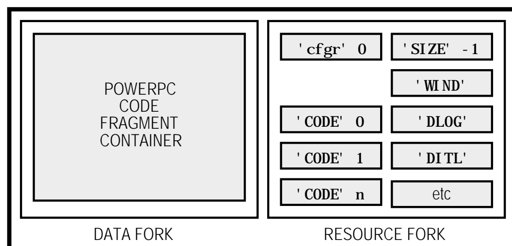


FIG 3 - THE STRUCTURE OFA FAT APPLICATION

Accelerated Resources

As previously stated, it is possible to use resources as containers for executable PowerPC code. You can put an executable PowerPC code fragment into a resource to obtain a PowerPC version of a 680x00 stand-alone code module. For example, you might re-compile an existing menu definition function (which is stored in a resource of type 'MDEF') into PowerPC code.

Note that, because the Menu Manager code that calls your menu definition function might be 680x0 code, a mode switch to the PowerPC environment might be required before your definition procedure can be executed. Accordingly, as shown at the left at Fig 4, you need to prepend a routine descriptor onto the beginning of the resource.

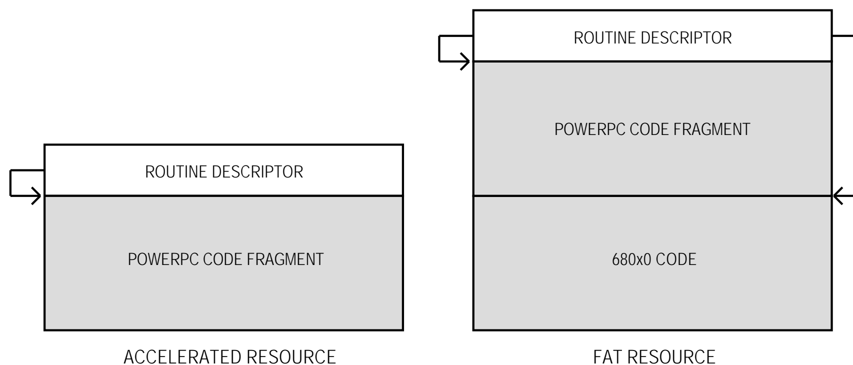


FIG 4 - STRUCTURE OF ACCELERATED AND FAT CODE-BEARING RESOURCES

These kinds of resources are called **accelerated resources** because they are faster implementations of existing kinds of resources. An accelerated resource is any resource containing PowerPC code that has a single entry point at the top (the routine descriptor) and that models the traditional behaviour of a 680x0 stand-alone code resource, for example:

- Menu definition functions (stored in a resources of type 'MDEF').
- Control definition functions (stored in a resources of type 'CDEF').

[5]Ideally, you should package your application as a fat application so as to afford users maximum flexibility. If, however, you decide not to package your application as a fat application, you should at least include an executable 680x0 'CODE' resource which displays an alert box informing the user that your application runs only on PowerPC-based Macintoshes.

- Window definition functions (stored in a resources of type `'WDEF'`).

You can transparently replace 680x0 code resources with accelerated PowerPC code resources without having to change the software (for example, an application) which uses them.

The creation of accelerated resources is described in the Codewarrior manual Targeting Mac OS at Chapter 5 (Creating Code Resource Projects)/Creating a Code Resource/Creating a PowerPC code resource project.

Fat Resources. As is shown at the right at Fig 4, it is also possible to create fat code-bearing resources, that is, resources containing both 680x0 and PowerPC versions of the same routine. In this case, the routine descriptor contains two routine records in its routineRecords field, one describing the 680x0 code and one describing the PowerPC code.[6]

Accelerated resources must not use global pointers (in C, pointers declared at `extern` or `static`) that are either initialised at run time or contained in dynamically allocated data structures to point to code or data in the resource itself. An accelerated resource can use uninitialised global data to point to objects in the heap. In addition, an accelerated resource can use global pointers that are initialised at compile time to point to functions, other global data, and literal strings, but these pointers cannot be modified at run time.

Calling Conventions

In the 680x0 environment, there are many ways for one routine to call another, depending on whether the called routine conforms to Pascal, C, Operating System, or other calling conventions. In the PowerPC environment, there is only one calling convention, which is designed to reduce the amount of time required to call another piece of code and to simplify the entire code calling process.

One significant feature of the new calling convention is that most parameters are passed in registers dedicated for that purpose. The large number of general-purpose and floating-point registers makes this goal quite easy to achieve. Parameters are passed on the stack only when they cannot be put into registers.

The Organisation of Memory

The organisation of memory in the PowerPC run-time environment is reasonably similar to the organisation of memory in the 680x0 run-time environment (see Chapter 1 — System Software, Memory, and Resources) in that:

- The system partition occupies the lowest memory address and most of the remaining space is allocated to the Process Manager, which creates a partition for each open application.

- The organisation of an application partition is reasonably similar to that for an application partition in the 680x0 run-time environment. In each application partition, there is a stack and a heap, as well as space for the application's global variables.

The two main differences between the 680x0 memory organisation and the PowerPC memory organisation concern the location of an application's code section and an application's global variables.

As previously stated, an application's executable code and global data are typically stored in a fragment container in the application's data fork. When the application is launched, the code and data sections of that fragment are loaded into memory. The data section is loaded into the application's heap. However, the location of the code section varies, depending on whether or not virtual memory is enabled.

---

[6]CodeWarrior does not create fat code resources automatically. Information on creating fat code resources is contained on the Reference CD at CodeWarrior Examples/Mac OS Examples/Code Resource Examples/Fat and Safe Code Resources.

Code Section Location - Virtual Memory Off

If virtual memory is not enabled, the code section of an application is loaded into the application heap. The Finder and Process Manager automatically expand your application partition as necessary to hold the code section. The code sections of other fragments are put into part of the Process Manager's heap known as **temporary memory**. If no temporary memory is available, code sections are loaded into the system heap. Fig 5 illustrates the general organisation of memory when virtual memory is not enabled.

Application partitions (including the application's stack, heap, and global variables) are loaded into the Process Manager heap. Code sections of applications and import libraries are loaded either into the Process Manager partition or (less commonly) into the system heap.

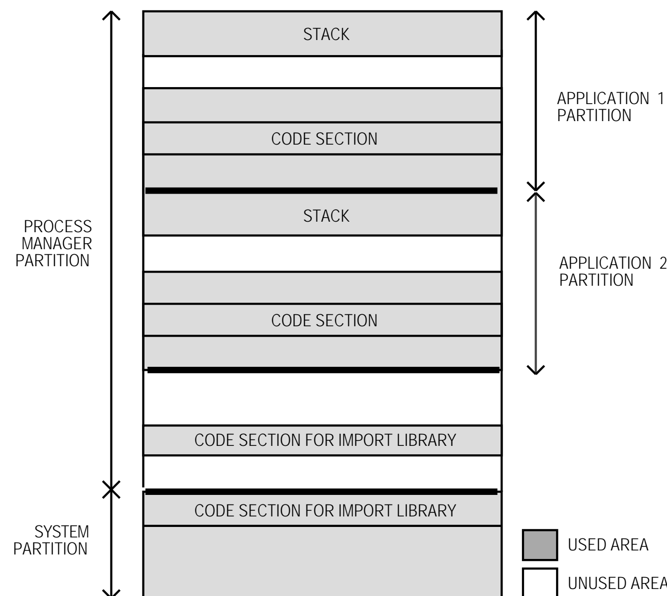


FIG 5 - ORGANIZATION OF MEMORY WHEN VIRTUAL MEMORY IS NOT ENABLED

Code Section Location - Virtual Memory On

If virtual memory is enabled, the Virtual Memory Manager uses a scheme called **file mapping** to map your application's fragment into memory. It uses the data fork of your application as the paging file for your application's code section. In the 680x0 environment, all unused pages of memory are written into a single system-wide backing-store file and re-read from there when needed. This often results in prolonged application launch, because an application's code is loaded into memory and sometimes immediately written out to the backing-store file. In the PowerPC environment, the entire code fragment is *mapped* into the logical address space, though only the needed portions of the code are actually *loaded* into physical memory.

Another benefit of the file mapping methodology is that, when it is time to remove some of your application's code from memory (to page other code and data in), the Virtual Memory Manager does not need to write the pages back to the paging file. Instead, it simply purges the code from the needed pages, because it can always read the file-mapped code back from the paging file (your application's data fork).

Fig 6 illustrates the general organisation of memory when virtual memory is enabled.

The virtual addresses occupied by the file-mapped pages of an application's (or an import library's) code are located outside both the system heap and the Process Manager's heap. As a result, an application's file-mapped code is never located in the application's heap itself.

Application partitions (including the application's heap, stack, and global variables) are loaded into the Process Manager heap, which is paged to and from the system-wide backing store file. Code sections and import libraries are paged directly from the data fork of the application or import library.

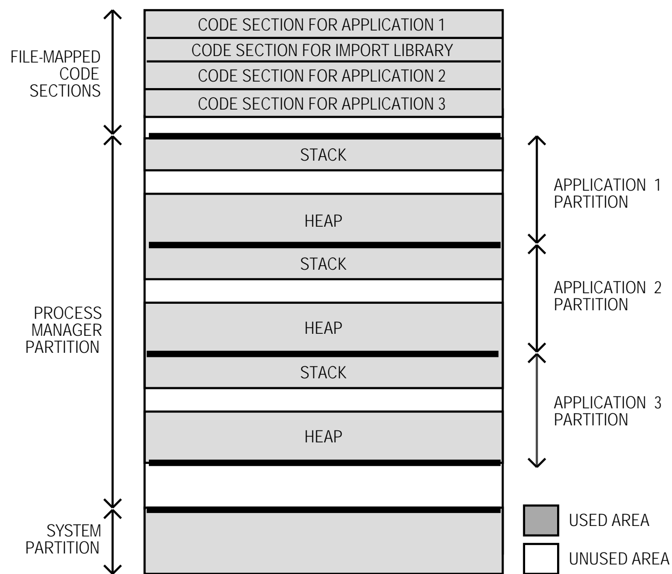


FIG 6 - ORGANIZATION OF MEMORY - VIRTUAL MEMORY ENABLED

Sometimes, however, parts of your application's executable code are loaded into your application's partition, not into the file-mapped space. This happens, for example, when you store an application extension (like a filter or a tool) as a resource in your application's resource fork. To make the code in that extension available, you need to call the Resource Manager to load it into your application's heap.

Structure of the System Partition

To support existing 680x0 applications and other software modules which access documented system global variables, the structure of much of the system partition remains unchanged.

Accessing System Global Variables

Note that the Universal Interfaces file LowMem.p contains declarations for routines that you can use to access virtually all of the documented system global variables. By using the routines provided by the system software, you can insulate your application or other software module from any future changes in the arrangement of low memory.

Structure of Application Partitions

The organisation of the application partition in the PowerPC environment is substantially simpler than in the 680x0 environment. However, the application partition for a PowerPC application (see Fig 7) consists only of a stack and a heap.

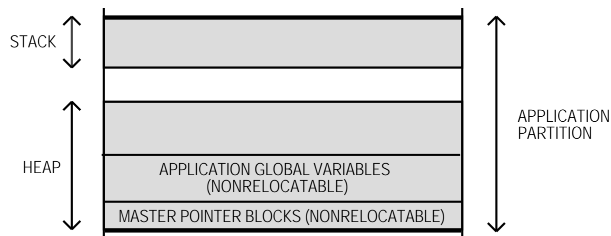


FIG 7 - STRUCTURE OF A POWERPC APPLICATION PARTITION

Demise of the A5 World

The A5 world which occupies part of a 680x0 application partition is largely absent from the PowerPC environment. The information maintained in the A5 world for 680x0 applications is either no longer needed by PowerPC applications or is maintained elsewhere (usually in the application heap). If your

application previously depended on some information being in its A5 world (that is, accesses it through the address in the A5 register), you will need to revise it to remove that dependence.

Recall from Chapter 1 — System Software, Memory, and Resources that the A5 world of a 680x0 application contains four kinds of data. The four kinds of data and their fate in the PowerPC environment, are as follows.

- **Jump Table.** A 680x0 application's jump table contains an entry for each of the application's routines that is called by code in another segment. Because the executable code in a PowerPC application is not segmented, there is no need for a jump table in a PowerPC application. Compilers which produce PowerPC code ignore any segmentation directives included in your source code, and any calls you make to the Segment Manager's `UnloadSeg` routine (see Chapter 22 — Miscellany) are simply ignored. In a PowerPC application, the task of keeping required code in memory is handled completely by the Virtual Memory Manager or the Process Manager, not by your application. Note that in CodeWarrior Pascal the routine `UnloadSeg` is undefined when compiling for PowerPC, requiring any calls to `UnloadSeg` to be conditionally compiled out or removed from PowerPC projects.

- **Application Global Variables.** In PowerPC applications, the application's global variables are part of the fragment's data section, which the Code Fragment Manager loads into the application's heap (see Fig 7). The application's global variables are always located in a single nonrelocatable block and are addressed through the fragment's **table of contents**[7].

- **Application Parameters.** The application parameters in a 680x0 application occupy 32 bytes, the first four bytes of which are a pointer to the application's QuickDraw global variables. In PowerPC applications, the application parameters are maintained privately by the Operating System.

- **QuickDraw Global Variables.** The QuickDraw global variables in a PowerPC application are stored as part of the application's global variables.

  Unless you have included the library MWCRuntime.Lib in your CodeWarrior PowerPC (and almost all PowerPC projects should include MWCRuntime.Lib), you will need to reserve space for the QuickDraw globals in your application and then pass the address to the `InitGraf` routine. If, in this circumstance, you want to maintain a single source code base for both the 680x0 and PowerPC environment, you can use conditional compilation, for example:

```
{$IFC GENERATINGCFM }
qd : QDGlobals;
{$ENDC}

procedure DoInitManagers;
  begin
  InitGraf(@qd.thePort);
  InitFonts;
  ...
  end;
```

  `GENERATINGCFM` is defined in the Universal Interfaces file ConditionalMacros.p.[8]

The Mini-A5 World

QuickDraw has been ported to native PowerPC code. However, even for applications which have themselves been ported to native PowerPC code, there must be a minimal A5 world to support some non-ported system software which accesses the QuickDraw global variables relative to the application's A5 value. This **mini-A5 world** contains nothing more than a pointer to the application's QuickDraw

[7]A fragment's table of contents contains pointers to the fragment's own static data. The table of contents also contains the addresses of code and data that the fragment imports, and which reside in some other fragment.

[8]The use of `GENERATINGCFM` (code being generated assumes CFM calling conventions), rather than `GENERATINGPOWERPC` (compiler is generating PowerPC instructions) arises from the fact that Apple has stated that the run-time environment defined by the use of fragments might in the future be available on 680x0-based Macintosh computers as well as PowerPC-based Macintosh computers. The new run-time environment based on fragments is intended to be as processor-independent as possible.

global variables which, as previously stated, reside in the application's global data section in PowerPC applications. The Process Manager creates a mini-A5 world for each native application at application launch time.

Accessing Global Variables From "Detached" Code

In the 680x0 environment, you need to manage your A5 explicitly when you need to gain access to your application's global variables or QuickDraw global variables from within some piece of "detached" code installed by your application, such as a VBL task. Recall the following from Chapter 19 — Custom Control Definition Functions and VBL Tasks:

> Because VBL tasks are interrupt routines, they could well execute when the value in the A5 register does not point to your application's A5 world. As a result, if you need to access your application's global variables in a VBL task, you need to set the A5 register to its correct value when your VBL task begins executing and restore the previous value upon exit.
>
> To achieve this, your application should save its A5 using `SetCurrentA5`. Then, at interrupt time, the VBL task can begin by calling `SetA5` to, firstly, set the A5 register to this saved value and, secondly, save the value that was in the A5 register immediately prior to the call. The VBL task should end with another call to `SetA5`, this time to restore the initial value.

All this is necessary in 680x0 code. However, because a PowerPC application does not have an A5 world, you never need to explicitly set up and restore your application's A5 world. (Put another way, your application's global variables are transparently available to any code compiled into your application.) Accordingly your VBL task source code needs to be modified for compilation as PowerPC code.

To maintain a single source code base for both the 680x0 and PowerPC environment, you can use conditional compilation. For example, consider the following simple 680x0 VBL task:

```
function GetVBLRec : VBLRecPtr;
  {$IFC NOT GENERATINGCFM}
  INLINE $2E88;                               { MOVE.L A0,D0 }
  {$ENDC}
...

procedure theVBLTask;
  var
  curA5 : Sint32;                             { For stored value of A5.}
  vblRecPtr : VBLRecPtr;                      { Pointer to task record.}
  ignoredLong : longint;

  begin
  taskRecPtr := GetVBLRec;                    { Get address of task record.}
  currentA5 := SetA5(vblRecPtr^.thisAppsA5);  { Set app's A5 and store old A5.}

  gCounter := gCounter + 1;                   { MODIFY A GLOBAL varIABLE.}
  taskRecPtr^.vblTaskRec.vblCount := kInterval; { Reset so function executes again.}

  ignoredLong := SetA5(currentA5);            { Restore the old A5 value.}
  end;
```

At the first call to `SetA5`, the application's A5 is set and the current value in the A5 register is saved. This saved value is re-installed at the second call to `SetA5`. For VBL tasks written as PowerPC code, both of these steps are unnecessary. Furthermore, in the 680x0 environment, the address of the VBL task record is passed in register A0. If you need that address in a high-level language, you must retrieve it immediately upon entry into your VBL task (as is done in the above listing). In the PowerPC environment, however, the address of the VBL task record can be passed to the task as an explicit parameter. The following shows how the above code may be modified for conditional compilation:

```
{$IFC GENERATING68K }
function GetVBLRec : VBLRecPtr;
  {$IFC NOT GENERATINGCFM}
  INLINE $2E88;                               { MOVE.L A0,D0 }
  {$ENDC}
{$ENDC}
...
```

```
{$IFC GENERATING68K }
procedure theVBLTask;
{$ELSEC}
procedure theVBLTask(vblRecPtr : VBLTaskPtr);
{$ENDC}
{
{$IFC GENERATING68K }
  var
  curA5 : longint;                              { For stored value of A5.}
  vblRecPtr : VBLRecPtr;                        { Pointer to task record.}
  ignoredLong : longint;
{$ENDC}

  begin

$IFC GENERATING68K }
  taskRecPtr := GetVBLRec;                      { Get address of task record.}
  currentA5 := SetA5(vblRecPtr^.thisAppsA5);    { Set app's A5 and store old A5.}
{$ENDC}

  gCounter := gCounter + 1;                     { MODIFY A GLOBAL varIABLE.}
  taskRecPtr^.vblTaskRec.vblCount := kInterval; { Reset so function executes again.}

{$IFC GENERATING68K }
  ignoredLong := SetA5(currentA5);              { Restore the old A5 value.}
{$ENDC}
  end;
```

GENERATING68K (compiler is generating 68K family instructions) is defined in the Universal Interfaces file ConditionalMacros.p.

Data Alignment

Unless told to do otherwise, a compiler arranges a data structure in memory so as to minimise the amount of time required to access the fields of the structure, which is generally what you would like to have happen. PowerPC and 680x0 compilers follow different conventions concerning the alignment of data in memory. Those conventions are as follows

- **680x0 Data Alignment Conventions.** A 680x0 processor places very few restrictions on the alignment of data in memory. The processor can read or write a byte, word, or long word value at any even address in memory. In addition, the processor can read byte values at any location in memory. As a result, the only padding required might be a single byte to align two-byte or larger fields to even boundaries or to make the size of an entire data structure an even number of bytes.

- **PowerPC Data Alignment Conventions.** The PowerPC processor *prefers* to access data in memory according to its natural alignment, which depends on the size of the data. A 1-byte value is always aligned in memory. A 2-byte value is aligned at an even address. A 4-byte value is aligned on any address that is divisible by 4, and so on. A PowerPC processor can access data that is *not* aligned on its natural boundary, but it performs aligned memory accesses more efficiently. As a result, PowerPC processors usually insert pad bytes into data structures to enforce the preferred data alignment. For example, consider this data structure, which would occupy 8 bytes in the 680x0 environment:

```
myStruct = record
  version : integer; {16-bits}
  address : longint; {32-bits}
  count : integer; {16-bits}
  end;
```

  To achieve the desired alignment of the address field in the PowerPC environment onto a 4-byte boundary, 2 bytes of padding are inserted after the version field. In addition, the structure itself is padded to a word boundary (a word on PowerPC processors being 4 bytes, not 2 bytes as is the case on 680x0 processors). As a result, the structure occupies 12 bytes of memory in the PowerPC environment.

In general, these different data alignment conventions should be transparent to your application. You need to worry about the differences only when you need to transfer data between the two environments, for example, when:

- Your application creates files containing data structures and the user copies those files from a PowerPC-based Macintosh to a 680x0-base Macintosh (or vice versa).
- Your PowerPC application creates a data structure and passes it to some code running under the 68LC040 Emulator.

To ensure that data can be transferred successfully in such cases, it is sufficient to simply instruct the PowerPC compiler to use 680x0 data alignment conventions. You can do this by choosing 68K in the Struct Alignment pop-up menu in the PPC Processor section of CodeWarrior's `Project Settings` dialog.

You should make sure, however, that you use 680x0 alignment only when absolutely necessary. The PowerPC processor is less efficient when accessing misaligned data.

Summary — Modifying 680x0 Code

In general, it is relatively easy to modify existing Pascal source code that successfully compiles and runs on 680x0-based Macintoshes so that it can be compiled and run on PowerPC-based Macintoshes. Most of the intricate work required to make your application compatible with the new PowerPC run-time environment is performed automatically by your development system's compiler and linker and by the Code Fragment Manager. The changes you need to make to your source code are summarised as follows:

- Create routine descriptors for any routines whose addresses you pass to code of an unknown type.
- Where there are dependencies on specific features of the 680x0 A5 world or the 680x0 run-time environment, modify your code for conditional compilation so as to exclude that dependency from the compiled PowerPC code.
- Where there are dependencies on information being passed in specific 680x0 registers, modify your code for conditional compilation so as to exclude that dependency from the compiled PowerPC code.
- Use 680x0 alignment for any data that is passed between environments.

In addition, you should minimise any dependencies on system global variables by using the new set of accessor routines defined in the Universal Interfaces file LowMem.p.

You also need to account for the fact that neither the PowerPC Floating-Point Unit nor the PowerPC numerics support Motorola's 80/96-bit `extended` type. This means that you will need to use Apple-supplied conversion utilities to move to and from `extended` (see below).

If you choose not to rebuild your application for the PowerPC environment, you should at least make certain that it does not violate any of the known limitations of the emulator.

Source Code Changes — Chapter 1 - 22 Demonstration Programs

The following source code changes are those required to:

- Enable the Macintosh Pascal demonstration programs to be compiled as either 680x0 code or PowerPC code.
- Enable the custom definition functions to be compiled as accelerated resources.

Routine Descriptors

Source Code File Controls2Pascal.p (Chapter 5)

In the global variables section, add:

```
actionProcedureRD : ControlActionUPP;                                   { For PowerPC }
```

Immediately after `DoInitManagers` in the main program block, add:

```
{ ................................................................ create routine descriptor }

actionProcedureRD := NewControlActionProc(ProcPtr(@ActionProcedure));   { For PowerPC }
```

In the `DoScrollBars` function, change the last call to `TrackControl` to:

```
then  ignored := TrackControl(controlHdl, mouseXY, actionProcedureRD)   { For PowerPC }
```

Source Code File DialogsAndAlertsPascal.p (Chapter 6)

In the global variables section, add:

```
eventFilterRD : ModalFilterUPP;                                         { For PowerPC }
drawDefaultButtonOutlineRD : UserItemUPP;                               { For PowerPC }
```

Immediately after `DoInitManagers` in the main program block, add:

```
{ ................................................................ create routine descriptors }

eventFilterRD := NewModalFilterProc(ProcPtr(@EventFilter));             { For PowerPC }
drawDefaultButtonOutlineRD := NewUserItemProc(ProcPtr(@DrawDefaultButtonOutline));
```

In the `DoDemonstrationMenu` function, change the call to `NoteAlert` to:

```
ignored := NoteAlert(rAlert, eventFilterRD);                            { For PowerPC }
```

In the `DoModalDialog` function, change the calls to `SetDialogItem` and `ModalDialog` to:

```
SetDialogItem(modalDlgPtr, iUserItem, itemType, Handle(drawDefaultButtonOutlineRD),
              itemRect);                                                { For PowerPC }

ModalDialog(eventFilterRD, itemHit);                                    { For PowerPC }
```

In the `DoMovableModal` function, change the call to `SetDialogItem` to:

```
SetDialogItem(modalDlgPtr, iUserItem, itemType, Handle(drawDefaultButtonOutlineRD),
              itemRect);                                                { For PowerPC }
```

In the `DoModelessDialog` function, change the call to `SetDialogItem` to:

```
SetDialogItem(gModelessDlgPtr, iUserItem, itemType,
              Handle(drawDefaultButtonOutlineRD), itemRect);            { For PowerPC }
```

Source Code File AppleEventsPascal.p (Chapter 8)

In the global variables section, add:

```
doOpenAppEventRD : AEEventHandlerUPP;                                   { For PowerPC }
doOpenDocsEventRD : AEEventHandlerUPP;                                  { For PowerPC }
doPrintDocsEventRD : AEEventHandlerUPP;                                 { For PowerPC }
doQuitAppEventRD : AEEventHandlerUPP;                                   { For PowerPC }
```

Immediately after `DoInitManagers` in the main program block, add:

```
{ ................................................................ create routine decriptors }

doOpenAppEventRD := NewAEEventHandlerProc(ProcPtr(@DoOpenAppEvent));     { For PowerPC }
doOpenDocsEventRD := NewAEEventHandlerProc(ProcPtr(@DoOpenDocsEvent));   { For PowerPC }
doPrintDocsEventRD:= NewAEEventHandlerProc(ProcPtr(@DoPrintDocsEvent));  { For PowerPC }
doQuitAppEventRD := NewAEEventHandlerProc(ProcPtr(@DoQuitAppEvent));     { For PowerPC }
```

Change the indicated lines in the function `DoInstallAEHandlers` as follows:

```
err := AEInstallEventHandler(kCoreEventClass, kAEOpenApplication,
    NewAEEventHandlerProc(AEEventHandlerProcPtr(@DoOpenAppEvent)), 0, false);
                                                                              { For PowerPC }
if (err <> noErr) then
   DoError(eInstallHandler);

err := AEInstallEventHandler(kCoreEventClass, kAEOpenDocuments,
    NewAEEventHandlerProc(AEEventHandlerProcPtr(@DoOpenDocsEvent)), 0, false);
                                                                              { For PowerPC }
if (err <> noErr) then
   DoError(eInstallHandler);

err := AEInstallEventHandler(kCoreEventClass, kAEPrintDocuments,
    NewAEEventHandlerProc(AEEventHandlerProcPtr(@DoPrintDocsEvent)), 0, false);
                                                                              { For PowerPC }
if (err <> noErr) then
  DoError(eInstallHandler);

err := AEInstallEventHandler(kCoreEventClass, kAEQuitApplication,
    NewAEEventHandlerProc(AEEventHandlerProcPtr(@DoQuitAppEvent)), 0, false);
                                                                              { For PowerPC }
if (err <> noErr) then
   DoError(eInstallHandler);
```

Source Code File FilesPascal.p (Chapter 14)

Make the same changes as for the source code file AppleEvents, but excluding code relating to the Print Documents event.

Source Code File Text1Pascal.p (Chapter 17)

In the global variables section add:

```
scrollActionProcRD : ControlActionUPP; external;                            { For PowerPC }
customClickLoopRD : TEClickLoopUPP; external;                               { For PowerPC }
```

Immediately after `DoInitManagers` in `main` function, add:

```
 { .......................................................... create routine descriptors }

scrollActionProcRD := NewControlActionProc(ProcPtr(@ScrollActionProc));     { For PowerPC }
customClickLoopRD := NewTEClickLoopProc(ProcPtr(@CustomClikLoop));          { For PowerPC }
```

Source Code File UText1Pascal.p (Chapter 17)

In the exported global variables section add:

```
scrollActionProcRD : ControlActionUPP;                                      { For PowerPC }
customClickLoopRD : TEClickLoopUPP;                                         { For PowerPC }
```

In the `DoInContent` function, change the first call to `TrackControl` to:

```
ignored := TrackControl(controlHdl, mouseXY, scrollActionProcRD);           { For PowerPC }
```

In the `DoNewDocWindow` function, change the call to `TESetClickLoop` to:

```
TESetClickLoop(customClickLoopRD, docRecHdl^^.editRecHdl);                  { For PowerPC }
```

Source Code File UHelpDialogPascal.p (Chapter 17)

In global variables section add:

```
helpDialogFilterRD : ModalFilterUPP;                                        { For PowerPC }
drawHelpRD : UserItemUPP;                                                   { For PowerPC }
actionProcedureRD : ControlActionUPP;                                       { For PowerPC }
```

At the bottom of the function interfaces section add:

```
procedure DisposeDescriptors; forward;                              { For PowerPC }
```

Immediately after the local variables in the `DoHelp` function, add:

```
helpDialogFilterRD := NewModalFilterProc(ProcPtr(@HelpDialogFilter));      { For PowerPC }
drawHelpRD := NewUserItemProc(ProcPtr(@DrawHelp));                         { For PowerPC }
actionProcedureRD := NewControlActionProc(ProcPtr(@ActionProcedure));      { For PowerPC }
```

Immediately after all calls to `DoErrorAlert` and `CloseHelp` in the `DoHelp` function, add:

```
DisposeDescriptors;                                                 { For PowerPC }
```

In the `DoHelp` function, change the call to `SetDialogItem` to:

```
SetDialogItem(modalDlgPtr, iTextUserItem, itemType,
              Handle(drawHelpRD), userItemRect);                    { For PowerPC }
```

In the `DoHelp` function, change the call to `ModalDialog` to:

```
ModalDialog(helpDialogFilterRD, itemHit);                           { For PowerPC }
```

In the `HandleScrollBar` function, change the call to `TrackControl` to:

```
ignored := TrackControl(docRecHdl^^.scrollbarHdl,mouseXY,actionProcedureRD);{ For PowerPC }
```

Add this new procedure:

```
{ ################################################################### DisposeDescriptors }

procedure DisposeDescriptors;

 begin
 DisposeRoutineDescriptor(helpDialogFilterRD);                      { For PowerPC }
 DisposeRoutineDescriptor(drawHelpRD);                              { For PowerPC }
 DisposeRoutineDescriptor(actionProcedureRD);                       { For PowerPC }
 end;
   {of procedure DisposeDescriptors}
```

Source Code File ListsPascal.p (Chapter 18)

In the global variables section, add:

```
doSearchPartialMatchRD : ListSearchUPP;                             { For PowerPC }
```

Immediately after `DoInitManagers` in the main program block, add:

```
 { ................................................................ create routine descriptors }

 doSearchPartialMatchRD := NewListSearchProc(ProcPtr(@DoSearchPartialMatch));
                                                                    { For PowerPC }
```

In the `DoTypeSelectSearch` function, change the call to `LSearch` to:

```
 if (LSearch(Ptr(longint(@gTSString) + 1), integer(gTSString[0]),
   doSearchPartialMatchRD, theCell, listHdl)) then                  { For PowerPC }
```

Source Code File CDEFandVBLPascal.p (Chapter 19)

In the global variables section, add:

```
animCursVBLTaskRD : VBLUPP;                                         { For PowerPC }
```

Immediately after `DoInitManagers` in the main program block, add:

```
{ ........................................................................ create routine descriptor }

animCursVBLTaskRD := NewVBLProc(ProcPtr(@AnimCursVBLTask));              { For PowerPC }
```

In the `DoInstallSystemVBLTask` procedure, change the third line to:

```
gVBLRec.vblTaskRec.vblAddr := animCursVBLTaskRD;                         { For PowerPC }
```

Source Code File CDEF2Pascal.p (Chapter 19)

Immediately after the type declaration block, add:

```
{ .................................................................................... global variables }

var

theVBLTaskRD : VBLUPP;                                                   { For PowerPC }
gVBLRec : VBLRec;                                                        { For PowerPC }
```

Immediately after the local variables in the `DoInitMessage` function, add:

```
 theVBLTaskRD := NewVBLProc(ProcPtr(@TheVBLTask));                       { For PowerPC }
```

In the `InstallVBLTask` function, change the fifth line to:

```
gVBLRec.vblTaskRec.vblAddr := theVBLTaskRD;                              { For PowerPC }
```

Immediately after the local variable in the `DoDisposeMessage` function, add:

```
DisposeRoutineDescriptor(theVBLTaskRD);                                  { For PowerPC }
```

Source Code File SoundPascal.p (Chapter 21)

In the global variables section, add:

```
drawDialogRD : UserItemUPP;                                              { For PowerPC }
```

Immediately after `doInitManagers` in the main program block, add:

```
{ ........................................................................ create routine descriptor }

drawDialogRD := NewUserItemProc(ProcPtr(@DrawDialog));                   { For PowerPC }
```

In the `DoSetUpDialog` function, change the call to `SetDialogItem` to:

```
SetDialogItem(gDialogPtr, iSynchSoundRect, itemType,
             Handle(drawDialogRD), itemRect);                            { For PowerPC }
```

Source Code File UDemos.p (Chapter 22)

In the global variables section, add:

```
doDeviceLoopDrawRD : DeviceLoopDrawingUPP;                               { For PowerPC }
```

Source Code File MiscellanyPascal.p (Chapter 22)

In the global variables section, add:

```
doDeviceLoopDrawRD : DeviceLoopDrawingUPP; external;                     { For PowerPC }
```

Immediately after `DoInitManagers` in the main program block, add:

```
{ ........................................................................ create routine descriptor }

doDeviceLoopDrawRD := NewDeviceLoopDrawingProc(ProcPtr(@DoDeviceLoopDraw)); { For PowerPC }
```

Source Code File UMain.p (Chapter 22)

In the global variables section, add:

```
doDeviceLoopDrawRD : DeviceLoopDrawingUPP; external;                    { For PowerPC }
```

In the `DoEvents` procedure in the unit UMain.p, change the call to `DeviceLoop` to:

```
DeviceLoop(theWindowPtr^.visRgn, doDeviceLoopDrawRD, userData, 0);      { For PowerPC }
```

Note: If the Chapter 22 notification demonstration had utilised a response procedure, a global variable of type `NMUPP` would have had to be declared, a routine descriptor would have had to be created by a call to `NewNMProc`, and the routine descriptor would have had to be assigned to the `nmResp` field of the Notification Record.

Conditional Compilation — A5 World Dependency (QuickDraw Globals)

All PowerPC projects for all Macintosh Pascal demonstration programs should include the library MWCRuntime.Lib. Accordingly, there is no requirement for any of the demonstration programs to explicitly reserve space for the QuickDraw globals, and there is thus no requirement for conditional compilation directives on that account.

Conditional Compilation — A5 World Dependency (VBL Tasks)

Source Code File CDEFandVBLPascal.p (Chapter 19)

Include conditional compilation directives as follows:

```
{ ........................................................................ in-line glue for GetVBLRec }

{$IFC GENERATING68K}                                                    { For PowerPC }
function GetVBLRec : longint;
 {$IFC NOT GENERATINGCFM}
 inline $2E88;
 {$ENDC}
{$ENDC}                                                                 { For PowerPC }

{ ################################################################ DoInstallSystemVBLTask }

procedure DoInstallSystemVBLTask;

 var
 ignored : OSErr;

 begin
 gVBLRec.vblTaskRec.qType := vType;
 gVBLRec.vblTaskRec.vblAddr := animCursVBLTaskRD;                       { For PowerPC }
 gVBLRec.vblTaskRec.vblCount := gVBLCount;
 gVBLRec.vblTaskRec.vblPhase := 0;

 {$IFC GENERATING68K}                                                   { For PowerPC }
 gVBLRec.thisApplicationsA5 := SetCurrentA5;
 {$ENDC}                                                                { For PowerPC }

 ignored := VInstall(QElemPtr(@gVBLRec.vblTaskRec));
 end;
   {of procedure DoInstallSystemVBLTask}

{ ####################################################################### AnimCursVBLTask }

procedure AnimCursVBLTask;

 {$IFC GENERATING68K}                                                   { For PowerPC }
 var
 theVBLRecPtr : VBLRecPtr;
 currentA5 : longint;
 {$ENDC}                                                                { For PowerPC }
```

```
begin
{$IFC GENERATING68K}                                            { For PowerPC }
theVBLRecPtr := VBLRecPtr(GetVBLRec);
currentA5 := SetA5(theVBLRecPtr^.thisApplicationsA5);
{$ENDC}                                                         { For PowerPC }

SetCursor(gAnimCursHdl^^.frame[gAnimCursHdl^^.whichFrame]^^);
gAnimCursHdl^^.whichFrame := gAnimCursHdl^^.whichFrame + 1;

if (gAnimCursHdl^^.whichFrame = gAnimCursHdl^^.numberOfFrames) then
  gAnimCursHdl^^.whichFrame := 0;

{$IFC GENERATING68K}                                            { For PowerPC }
theVBLRecPtr^.vblTaskRec.vblCount := gVBLCount;
{$ELSEC}                                                        { For PowerPC }
gVBLRec.vblTaskRec.vblCount := gVBLCount;                       { For PowerPC }
{$ENDC}                                                         { For PowerPC }

{$IFC GENERATING68K}                                            { For PowerPC }
currentA5 := SetA5(currentA5);
{$ENDC}                                                         { For PowerPC }
end;
  {of procedure AnimCursVBLTask}
```

Source Code File CDEF2Pascal.p (Chapter 19)

Include conditional compilation directives as follows:

```
{ ............................................................ in-line glue for GetVBLRec }

{$IFC GENERATING68K}                                             { For PowerPC }
function GetVBLRec : longint;
 {$IFC NOT GENERATINGCFM}
 INLINE $2E88;
 {$ENDC}
{$ENDC}                                                          { For PowerPC }

{ ######################################################################## InstallVBLTask }

function InstallVBLTask(theControl : ControlHandle) : OSErr;

 var
 theErr : OSErr;
 theSliderDataHdl : SliderDataHdl;

 begin
 theSliderDataHdl := SliderDataHdl(theControl^^.contrlData);
 gVBLRec.inVBlankPeriod := false;

 gVBLRec.vblTaskRec.qType := vType;
 gVBLRec.vblTaskRec.vblAddr := theVBLTaskRD;                     { For PowerPC }
 gVBLRec.vblTaskRec.vblCount := 1;
 gVBLRec.vblTaskRec.vblPhase := 0;

 {$IFC GENERATING68K}                                            { For PowerPC }
 gVBLRec.thisApplicationsA5 := SetCurrentA5;
 {$ENDC}                                                         { For PowerPC }

 if (theSliderDataHdl^^.slotVInstallPresent) then
   theErr := SlotVInstall(QElemPtr(@gVBLRec.vblTaskRec),
               theSliderDataHdl^^.mainSlotNumber)
 else
   theErr := VInstall(QElemPtr(@gVBLRec.vblTaskRec));

 InstallVBLTask := theErr;
 end;
   {of function InstallVBLTask}

{ ############################################################################ TheVBLTask }

procedure TheVBLTask;

 {$IFC GENERATING68K}                                            { For PowerPC }
 var
 theVBLRecPtr : VBLRecPtr;
 currentA5 : longint;
```

```
ignoredLong : longint;
{$ENDC}                                                              { For PowerPC }

begin
{$IFC GENERATING68K}                                                 { For PowerPC }
theVBLRecPtr := VBLRecPtr(GetVBLRec);
currentA5 := SetA5(theVBLRecPtr^.thisApplicationsA5);
{$ENDC}                                                              { For PowerPC }

{$IFC GENERATING68K}                                                 { For PowerPC }
theVBLRecPtr^.inVBlankPeriod := true;
theVBLRecPtr^.vblTaskRec.vblCount := 1;
{$ELSEC}                                                             { For PowerPC }
gVBLRec.inVBlankPeriod := true;                                      { For PowerPC }
gVBLRec.vblTaskRec.vblCount := 1;                                    { For PowerPC }
{$ENDC}                                                              { For PowerPC }

{$IFC GENERATING68K}                                                 { For PowerPC }
ignoredLong := SetA5(currentA5);
{$ENDC}                                                              { For PowerPC }
end;
  {of function TheVBLTask}
```

Conditional Compilation — A4 Register Dependency

Include the the following conditional compilation directives in the `main` functions in the source code files for the two custom CDEFs at Chapter 19 and the custom WDEF at Chapter 20:

```
{$IFC GENERATING68k}                                                 { For PowerPC }
oldA4, ignored : longint;
{$ENDC}                                                              { For PowerPC }

{$IFC GENERATING68K }                                                { For PowerPC }
oldA4 := SetCurrentA4;
{$ENDC}                                                              { For PowerPC }

{$IFC GENERATING68K }                                                { For PowerPC }
ignored := SetA4(oldA4);
{$ENDC}                                                              { For PowerPC }
```

Conditional Compilation — Code Segmentation

In CodeWarrior Pascal, the procedure `UnloagSeg` is undefined when compiling for PowerPC (see the comments in the Universal Interface file SegLoad.p), so any calls to `UnloagSeg` should be compiled out. Thus in Chapter 22 (Miscellany), the `UnloagSegments` procedure needs to have the following conditional compilation directives added:

```
{ ######################################################################## UnloadSegments }

procedure UnloadSegments;

 begin
 {$IFC GENERATING68K }                                               { For PowerPC }
 UnloadSeg(@DemosSegment);
 {$ENDC}                                                             { For PowerPC }
 end;
   {of procedure UnloadSegments}
```

Accessor Routines for System Global Variables

No changes are required because the correct accessor routines are already used in all the source code files, specifically:

- `LMGetCaretTime` in the source code files DialogsAndAlertsPascal.p, Text1Pascal.p, and Text2Pascal.p.
- `LMGetHiliteRGB`, `LMGetHiliteMode`, and `LMSetHiliteMode` in the source code files ColorQuickDrawPascal.p (Chapter 11) and ListsPascal.p (Chapter 18).

- `LMGetKeyThresh` in the source code file ListsPascal.p (Chapter 18).
- `LMGetWindowList` and `LMSetWindowList` in the source code file FloatRoutinesPascal.p (Chapter 20).
- `LMGetMainDevice` in the source code files GDevicePascal.p (Chapter 9), ColorQuickDrawPascal.p (Chapter 11), WDEFPascal.p (Chapter 19), and UDemos.p (Chapter 22).
- `LMGetEventQueue`, `LMGetDeviceList` and `LMGetMBarHeight` in the source code file UDemos.p (Chapter 22).

Floating Point Arithmetic

The PowerPC-based Macintosh follows the IEEE 754 standard for floating-point arithmetic. In this standard, `float` is 32 bits, and `double` is 64 bits. (Apple has added a 128 bit `long double` type.) However, the PowerPC Floating-Point Unit does not support Motorola's 80/96-bit `extended` type, and neither do the PowerPC numerics. To accommodate this, you can use Apple-supplied conversion utilities to move to and from `extended`.

If the source code file Text2Pascal.p (Chapter 17), include the Universal Interfaces file fp.p, the import library MathLib and, in the procedure `DoAcceptValueField`:

- Add the following local variable:

```
valueDouble : Double;                                        { For PowerPC }
```

- Replace the line `value80Bit := value80Bit * quantity;` with the following:

```
valueDouble := x80tod(value80Bit);                           { For PowerPC }
valueDouble := valueDouble * quantity;                       { For PowerPC }
dtox80(valueDouble, value80Bit);                             { For PowerPC }
```

Relevant Constants, Data Types, Routines and Definitions

Contants

Instruction Set Architectures

```
kM68kISA          = 0  { 680x0 architecture. }
kPowerPCISA       = 1  { PowerPC architecture. }
```

Procedure Information

```
kPascalStackBased   = 0
kCStackBased        = 1
kRegisterBased      = 2
```

Data Types

```
type
  ISAType = SInt8;
  CallingConventionType = integer;
  ProcInfoType = longint;
```

Routine Descriptor

```
type
  RoutineDescriptor = packed record
  goMixedModeTrap:  UInt16;             { Our A-Trap }
  version:          SInt8;              { Current Routine Descriptor version }
  routineDescriptorFlags: RDFlagsType;  { Routine Descriptor Flags }
  reserved1:        UInt32;             { Unused, must be zero }
  reserved2:        Uint8;              { Unused, must be zero }
  selectorInfo:     Uint8               { If a dispatched routine, calling convention, else 0 }
```

```
  routineCount:     UInt16;             { Number of routines in this RD }
  routineRecords:   array [0..0] of RoutineRecord;      { The individual routines }
  end;

  RoutineDescriptorPtr = ^RoutineDescriptor;
  RoutineDescriptorHandle = ^RoutineDescriptorPtr;
```

Routine Record

```
type
  RoutineRecord = record
  procInfo:         ProcInfoType;       { calling conventions }
  reserved1:        SInt8; (* UInt8 *)  { Must be 0 }
  ISA:              ISAType;            { Instruction Set Architecture }
  routineFlags:     RoutineFlagsType;   { Flags for each routine }
  procDescriptor:   ProcPtr;            { Where is the thing we're calling? }
  reserved2:        UInt32;             { Must be 0 }
  selector:         UInt32;             { For dispatched routines, the selector }
  end;

  RoutineRecordPtr = ^RoutineRecord;
  RoutineRecordHandle = ^RoutineRecordPtr;
```

Type Definitions Relevant to Routine Descriptors

```
type

ProcPtr = Ptr;
Register68kProcPtr = ProcPtr;  { procedure ; }
ProcHandle = ^ProcPtr;
UniversalProcPtr = ProcPtr;
UniversalProcHandle = ^ProcPtr;

AEEventHandlerUPP = UniversalProcPtr;
ControlActionUPP = UniversalProcPtr;
ControlDefUPP = UniversalProcPtr;
DeviceLoopDrawingUPP = UniversalProcPtr;
ListDefUPP = UniversalProcPtr;
ListSearchUPP = UniversalProcPtr;
MenuDefUPP = UniversalProcPtr;
ModalFilterUPP = UniversalProcPtr;
TEClickLoopUPP = UniversalProcPtr;
UserItemUPP = UniversalProcPtr;
VBLUPP = UniversalProcPtr;
WindowDefUPP = UniversalProcPtr;

{$IFC PROCTYPE }
AEEventHandlerProcPtr = function (var theAppleEvent: AppleEvent; var reply: AppleEvent;
       handlerRefcon: longint): OSErr;
ControlActionProcPtr = procedure (theControl: ControlRef; partCode: ControlPartCode);
ControlDefProcPtr = function (varCode: SInt16; theControl: ControlRef;
       message: ControlDefProcMessage; param: SInt32): SInt32;
DeviceLoopDrawingProcPtr = procedure (depth: integer; deviceFlags: integer;
       targetDevice: GDHandle; userData: longint);
ListDefProcPtr = procedure (lMessage: integer; lSelect: boolean; var lRect: Rect; lCell: Cell;
       lDataOffset: integer; lDataLen: integer; lHandle: ListRef);
ListSearchProcPtr = function (aPtr: Ptr; bPtr: Ptr; aLen: integer; bLen: integer): integer;
MenuDefProcPtr = procedure (message: integer; theMenu: MenuRef; var menuRect: Rect;
       hitPt: Point; var whichItem: integer);
ModalFilterProcPtr = function (theDialog: DialogPtr; var theEvent: EventRecord;
       var itemHit: integer): boolean;
UserItemProcPtr = procedure (theWindow: WindowPtr; itemNo: integer);
WindowDefProcPtr = function (varCode: integer; theWindow: WindowRef; message: integer;
       param: longint): longint;
{$ELSEC}
AEEventHandlerProcPtr = ProcPtr;
ControlActionProcPtr = ProcPtr;
ControlDefProcPtr = ProcPtr;
DeviceLoopDrawingProcPtr = ProcPtr;
ListDefProcPtr = ProcPtr;
ListSearchProcPtr = ProcPtr;
MenuDefProcPtr = ProcPtr;
ModalFilterProcPtr = ProcPtr;
```

```
UserItemProcPtr = ProcPtr;
WindowDefProcPtr = ProcPtr;
{$ENDC}
```

Routines

Calling Routines Via Universal Procedure Pointers

```
function CallUniversalProc(theProcPtr: UniversalProcPtr; procInfo: ProcInfoType; ...):
          longint; C; external;
```

Determining Instruction Set Architectures

```
{$IFC GENERATINGPOWERPC }
        GetCurrentISA       = kPowerPCISA;
        GetCurrentRTA       = kPowerPCRTA;

{$ELSEC}
    {$IFC GENERATINGCFM }
        GetCurrentISA       = kM68kISA;
        GetCurrentRTA       = kCFM68kRTA;

    {$ELSEC}
        GetCurrentISA       = kM68kISA;
        GetCurrentRTA       = kOld68kRTA;

    {$ENDC}
{$ENDC}
        GetCurrentArchitecture   = 0+(GetCurrentISA + GetCurrentRTA);
```

Creating and Disposing of Routine Descriptors

```
function  NewRoutineDescriptor(theProc: ProcPtr; theProcInfo: ProcInfoType;
          theISA: ByteParameter): UniversalProcPtr;

function  NewFatRoutineDescriptor(theM68kProc: ProcPtr; thePowerPCProc: ProcPtr;
          theProcInfo: ProcInfoType): UniversalProcPtr;

procedure DisposeRoutineDescriptor(theProcPtr: UniversalProcPtr);

function NewControlActionProc(userRoutine: ControlActionProcPtr): ControlActionUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewControlDefProc(userRoutine: ControlDefProcPtr): ControlDefUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewModalFilterProc(userRoutine: ModalFilterProcPtr): ModalFilterUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewUserItemProc(userRoutine: UserItemProcPtr): UserItemUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewListDefProc(userRoutine: ListDefProcPtr): ListDefUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewWindowDefProc(userRoutine: WindowDefProcPtr): WindowDefUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewMenuDefProc(userRoutine: MenuDefProcPtr): MenuDefUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}
```

```
function NewTEClickLoopProc(userRoutine: TEClickLoopProcPtr): TEClickLoopUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewAEEventHandlerProc(userRoutine: AEEventHandlerProcPtr): AEEventHandlerUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewListSearchProc(userRoutine: ListSearchProcPtr): ListSearchUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewDeviceLoopDrawingProc(userRoutine: DeviceLoopDrawingProcPtr):
         DeviceLoopDrawingUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}

function NewVBLProc(userRoutine: VBLProcPtr): VBLUPP;
         {$IFC NOT GENERATINGCFM }
         INLINE $2E9F;
         {$ENDC}
```

Procedure Information Definitions

```
const

uppControlActionProcInfo = $000002C0;
{ procedure (4 byte param, 2 byte param); }

uppControlDefProcInfo = $00003BB0;
{ function (2 byte param, 4 byte param, 2 byte param, 4 byte param): 4 byte result; }

uppListDefProcInfo = $000EBD80;
{ procedure (2 byte param, 1 byte param, 4 byte param, 4 byte param, 2 byte param,
     2 byte param, 4 byte param); }

uppMenuDefProcInfo = $0000FF80;
{ procedure (2 byte param, 4 byte param, 4 byte param, 4 byte param, 4 byte param); }

uppTEClickLoopProcInfo = $0000F812;
{ Register function (4 bytes in A3): 1 byte in D0; }

uppWindowDefProcInfo = $00003BB0;
{ function (2 byte param, 4 byte param, 2 byte param, 4 byte param): 4 byte result; }

uppAEEventHandlerProcInfo = $00000FE0;
{ function (4 byte param, 4 byte param, 4 byte param): 2 byte result; }

uppModalFilterProcInfo = $00000FD0;
{ function (4 byte param, 4 byte param, 4 byte param): 1 byte result; }

uppUserItemProcInfo = $000002C0;
{ procedure (4 byte param, 2 byte param); }

uppListSearchProcInfo = $00002BE0;
{ function (4 byte param, 4 byte param, 2 byte param, 2 byte param): 2 byte result; }

uppDeviceLoopDrawingProcInfo = $00003E80;
{ procedure (2 byte param, 2 byte param, 4 byte param, 4 byte param); }

uppVBLProcInfo = $00009802;
{ Register procedure (4 bytes in A0); }
```