5

MICROPROCESSOR MATTERS

Introduction

The demonstration programs at Chapter 7 — Introduction to Controls are the first in which specific measures have had to be taken to ensuring that the source code will compile satisfactorily as both 680x0 code and PowerPC code. Accordingly, as a prelude for what is to come, this chapter addresses the differing run time environments[1] of the 680x0 and PowerPC microprocessors and the measures required to ensure that source code that will compile successfully as both 680x0 and PowerPC code.

The 68LC040 Emulator

The system software provides the ability to execute applications that use the native instruction set of the Power Macintosh's PowerPC microprocessor *and* applications which use the Motorola 680x0 instruction set. The ability to execute applications which use the 680x0 instruction set is provided by an **emulator** (the 68LC040 Emulator), which provides an execution environment that 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.

[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.

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.

An important aspect of the emulator is that it 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 Mixed Mode Manager

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)**, 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 function (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 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.

- When a PowerPC application invokes a system software function 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 function 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 an **action function**[2] 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 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:

[2] Action functions (sometimes called **hook functions** or **call-back functions**) refer to the ability of a Toolbox function to itself call an application-defined function during its execution, thus extending the features of the function. Action functions are used for the first time in the demonstration programs associated with Chapter 7 — Controls. That is the main reason why the demonstration programs at Chapter 7 are the first in which specific measures have had to be taken to ensuring that the source code will compile satisfactorily as both 680x0 code and PowerPC code.

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

An Example - Control Action Functions

As an example, suppose you are writing a native PowerPC application that calls the Control Manager function TrackControl. TrackControl accepts as one of its parameters the address of an action function 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 function, so you cannot know whether your action function and the TrackControl function are of the same ISA.

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

Indicating the ISA of a Callback Function — Routine Descriptors

You indicate the ISA of a particular function by creating a **routine descriptor** for that function (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 function. In the simplest case, the array of routine records contains a single element.

```
TYPE
  RoutineDescriptorPtr = ^RoutineDescriptor;
  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;

RoutineDescriptorHandle = ^RoutineDescriptorPtr;
```

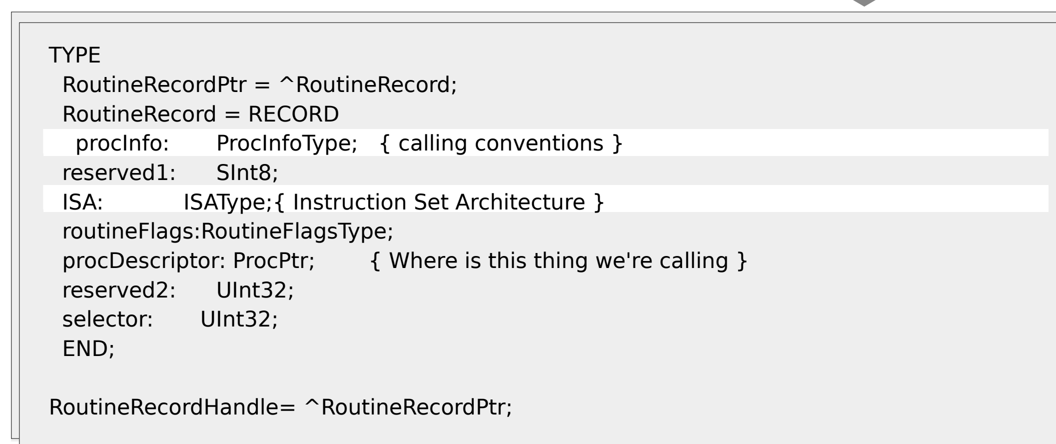

FIG 1 -THE ROUTINE DESCRIPTOR RECORD AND ROUTINE RECORD

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 function being described. It always contains one of these two constants, which are defined in the Universal Interfaces file MixedMode.p:

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

- ***procInfo Field.*** The procInfo field contains the function's **function information**, which encodes the function's calling conventions and information about the number and location of the function's parameters. For the standard kinds of action functions and other types of "detached" code, the Universal Interfaces files include definitions of function information. For example, the interface file Controls.p includes this definition (the comment has been added to explain how the value of the "magic constant" has been calculated):

```
CONST
   uppControlActionProcInfo = $000002C0;

{ $000002C0 = kPascalStackBased + kNoByteCode*$10 + kFourByteCode*$40 +
         kTwoByteCode*$100 }
```

  This function information specification indicates that a control action function follows standard Pascal calling conventions and takes two stack-based parameters (a control handle and a part code) and returns no result. Note that when calculating the value of the procInfo the codes for the arguments and return value are multiplied by constants. This is equivalent to shifting the bits in the kByteCode constants to the left so that the length of the return value is specified in bits 5-6, the length of the first parameter in bits 7-8 and the length of the second parameter in bits 9-10.

Creating a Routine Descriptor For a Control Action Function

The Universal Interfaces file Controls.p contains this definition for the NewControlActionProc function:

```
type
ControlActionUPP : UniversalProcPtr;

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

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

```
myControlActionUPP : ControlActionUPP;

myControlActionUPP := NewControlActionProc(ControlActionProcPtr(@myControlAction));
```

Notice that the result returned by NewControlActionProc is of type ControlActionUPP. The UPP stands for a **universal procedure pointer**, which is defined in the Universal Interfaces to be either a pointer to a routine descriptor or a simple 680x0 function pointer (hence the term "universal"). Thus the effect of the call to NewControlActionProc depends on whether it is executed in the 680x0 environment or the PowerPC environment:

- In the 680x0 environment, NewControlActionProc simply returns its first parameter, that is, a pointer to your application-defined control action function.

- In the PowerPC environment, NewControlActionProc 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);
```

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

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.

Functions Requiring Routine Descriptors

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

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

- Control action functions (see the demonstration programs at Chapter 7 — Introduction to Controls and Chapter 19 — Text and TextEdit).
- Event filter functions (see the demonstration program at Chapter 8 — Dialogs and Alerts and Chapter 19 — Text and TextEdit).
- Apple event handling functions (see the demonstration programs at Chapter 10 — Required Apple Events and Chapter 16 — Files).
- TextEdit click loop functions (see the first demonstration program at Chapter 19 — Text and TextEdit).

The PowerPC Environment

In the emulation environment provided by the 68LC040 emulator, the organisation of an application partition[3], and the run-time behaviour of emulated software are identical to that provided on 680x0-based Macintoshes. However, the run-time environment 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

[3] See Chapter 1 — System Software, Memory, and Resources.

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.[4] 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[5]. In the PowerPC environment, any function 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[6].

---

[4] 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.

[5] 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.

[6] 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.

Container Formats

The Code Fragment Loader recognises two kinds of container format, the principal one being **Preferred Executable Format** (**PEF**). PEF is an object file format developed by Apple Computer. 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.

You do not have to create the 'cfrg' resource yourself because CodeWarrior does this for you when you compile your application as PowerPC code.

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).

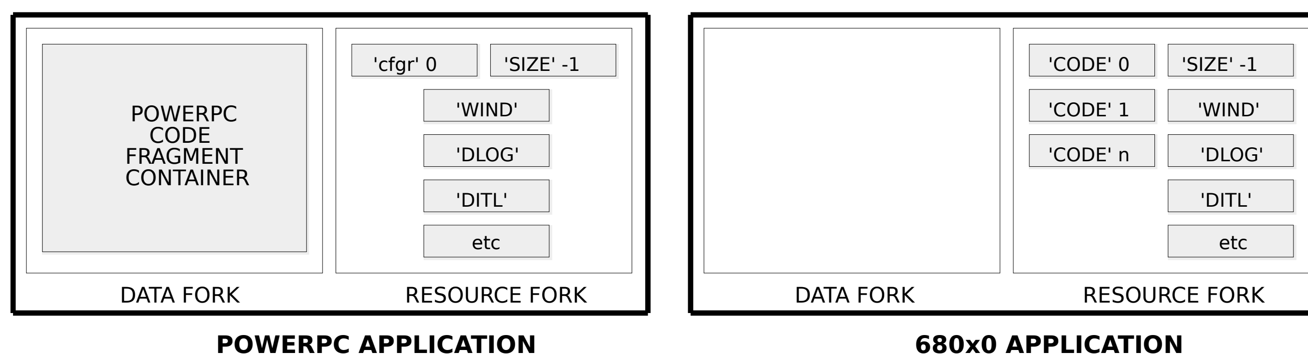

FIG 2 - THE STRUCTURE OF 680x0 AND POWERPC APPLICATIONS

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.[7]

[7] 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.

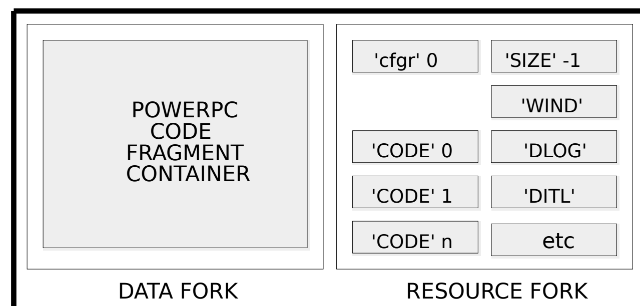

FIG 3 - THE STRUCTURE OFA FAT APPLICA

Accelerated Code Resources

As previously stated, it is possible to use resources as containers for executable code. You can put an executable PowerPC code fragment into a resource to obtain a PowerPC version of a 680x0 stand-alone code module.

For example, you might re-compile an existing custom menu definition function (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 function 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.

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, a control definition function (stored in a resource of type 'CDEF').

The CodeWarrior Linker adds the start-up code to your resource that allows a 680x0 application, running under the emulator on a PowerPC, to use the resource. The resource runs only on a PowerPC computer.

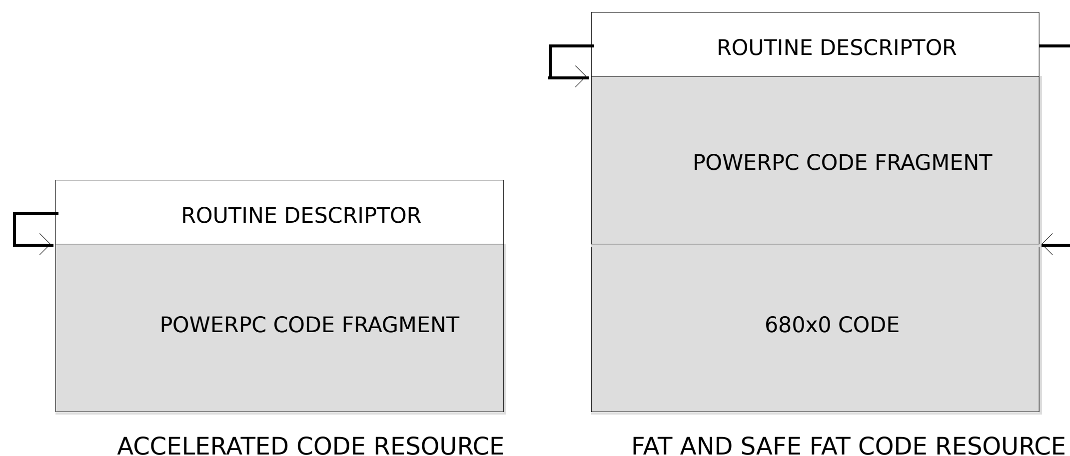

FIG 4 - STRUCTURE OF ACCELERATED AND FAT CODE RESOURCES

Fat Code Resources

As is shown at the right at Fig 4, it is also possible to create **fat code resources**, that is, resources containing both 680x0 and PowerPC versions. A fat code resource runs only on a PowerPC computer. It contains a header that allows the Mixed Mode Manager to avoid a context switch by letting the resource run either native or under the 68K emulator.

Safe Fat Code Resources

You can also create **safe fat code resources**, which run on either 680x0 or PowerPC. A safe fat resource contains a header that decides whether the resource is running on a 680x0 or a PowerPC, and calls the correct code for the Mac OS computer that it is running on.

Calling Conventions

In the 680x0 environment, there are many ways for one function to call another, depending on whether the called function 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 calling convention in the PowerPC environment is that most parameters are passed in registers dedicated for that purpose. The large number of general-purpose and floating-point registers in the PowerPC makes this goal quite easy to achieve. Parameters are passed on the stack only when they cannot be put into registers.

Accessing Global Variables — "Detached" Code

In the 680x0 environment, you need to manage the processor's A5 register 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. As will be seen at Chapter 23 — Miscellany, an example of such a piece of detached code is a VBL task (vertical blanking task).

Because VBL tasks are interrupt routines, they could well execute when the value in the 680x0 microprocessor's A5 register does not point to your application's A5 world. As a result, if you needed to access your application's global variables within a VBL task, you would 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 would save its A5 using SetCurrentA5. Then, at interrupt time, the VBL task would 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 would then 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 (see Chapter 1 — System Software, Memory, and Resources), 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 would need to be modified for compilation as PowerPC code.

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

```
function GetVBLRec : VBLRecPtr;
   inline $2E88;  { MOVE.L A0,D0 }

...

procedure TheVBLTask;

   var
   ignoredInt : SInt32;
   currentA5 : SInt32;                              { For stored value of A5. }
   taskRecPtr: VBLRecPtr;                           { Pointer to task record. }
```

```
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. }

ignoredInt := SetA5(currentA5);                      { Restore the old A5 value. }
end;
    { of procedure TheVBLTask }
```

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 source code would be modified for conditional compilation so as to be capable of being compiled as both 680x0 code and PowerPC code:

```
{$ifc TARGET_CPU_68K }
function GetVBLRec : VBLRecPtr;
    inline $2E88; { MOVE.L A0,D0 }
{$endc}
...
{$ifc TARGET_CPU_68K }
procedure TheVBLTask;
{$elsec}
procedure TheVBLTask(vblRecPtr : VBLTaskPtr);
{$endc}

    var
    taskRecPtr: VBLRecPtr;                              { Pointer to task record. }
{$ifc TARGET_CPU_68K }
    ignoredSInt32 : SInt32;
    curA5 : SInt32;                                     { For stored value of A5. }

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

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

{$ifc TARGET_CPU_68K }
    ignoredSInt32 := SetA5(currentA5);                  { Restore the old A5 value. }
{$endc}
    end;
        { of procedure TheVBLTask }
```

TARGET_CPU_68K (Compiler is generating 680x0 instructions) is defined in the Universal Interfaces file ConditionalMacros.p.

Accessing Global Variables — Code Resources

Code resources (for example, custom menu definition functions) often need to access global variables. To avoid any conflict with the running application, 680x0 versions of code resources access their globals referenced from the 680x0 A4 register, instead of the A5 register. Accordingly, the value in the A4 register must be saved on entry to the main function and restored on exit. Using CodeWarrior, you would ensure this by calling SetCurrentA4 and SetA4 as follows:

```
uses
    PascalA4;
```

```
var
oldA4, ignored : longint;

begin
    oldA4 := SetCurrentA4;

    ...

    ignored := SetA4(oldA4);
end.
```

The calls to EnterCodeResource and ExitCodeResource are not required when the code is being compiled as PowerPC code. Accordingly the above code would need to be modified for conditional compilation so as to be capable of being compiled as both 680x0 code and PowerPC code:

```
uses
    PascalA4;

{$ifc TARGET_CPU_68K }
var
oldA4, ignored : longint;
{$endc}

begin
{$ifc TARGET_CPU_68K }
    oldA4 := SetCurrentA4;
{$endc}

    ...

{$ifc TARGET_CPU_68K }
    ignored := SetA4(oldA4);
{$endc}
end.
```

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. The 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 eight bytes in the 680x0 environment:

  ```
  MyRecord = record
      version : SInt16;
      address : SInt32;
      count : SInt16;
      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 record 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 record 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 in the CodeWarrior 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.

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 must use Apple-supplied conversion utilities to move to and from `extended`. Once again, conditional compilation may be used to make your code 680x0/PowerPC compatible. The following is an example:

```
quantity : Sint32;
value80bit : extended80;
{$ifc TARGET_CPU_PPC }
valueDouble : double;
{$endc}

{$ifc TARGET_CPU_68K }
value80bit := value80Bit * quantity;
{$endc}
{$ifc TARGET_CPU_PPC }
    valueDouble := x80tod(value80Bit);
    valueDouble := valueDouble * quantity;
    dtox80(valueDouble, value80Bit);
{$endc}
```

Making Code Suitable For Compilation As Either 680x0 Code or PowerPC code — Summary

In general, it is relatively easy to ensure that your code will compile successfully as either 680x0 code or PowerPC code. The areas requiring attention 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, use 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 (for example the A4 register), use conditional compilation so as to exclude that dependency from the compiled PowerPC code.

- Use 680x0 alignment for any data that is passed between environments.
- Accommodate the fact that neither the PowerPC Floating-Point Unit nor the PowerPC numerics support Motorola's 80/96-bit extended type.

Relevant Constants, Data Types, and Functions

In the following, those functions introduced with Mac OS 8 and the Appearance Manager are shown on a light gray background.

Constants

Instruction Set Architectures

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

Procedure Information

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

Data Types

```
ISAType = SInt8;
CallingConventionType = INTEGER;
ProcInfoType = LONGINT;
```

Routine Descriptor

```
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

```
RoutineRecord = RECORD
   procInfo:          ProcInfoType;         {  Calling conventions  }
   reserved1:         SInt8;                {  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;
```

Routines

Determining Instruction Set Architectures

```
FUNCTION  GetCurrentISA : ISAType;  { Actually a conditionally-defined constant. }
```

Creating and Disposing of Routine Descriptors

```
FUNCTION  NewRoutineDescriptor(theProc: ProcPtr; theProcInfo: ProcInfoType;
                theISA: ISAType): UniversalProcPtr;

FUNCTION  NewFatRoutineDescriptor(theM68kProc: ProcPtr; thePowerPCProc: ProcPtr;
                theProcInfo: ProcInfoType): UniversalProcPtr;
PROCEDURE DisposeRoutineDescriptor(theProcPtr: UniversalProcPtr);

FUNCTION NewControlActionProc(userRoutine: ControlActionProcPtr): ControlActionUPP;
FUNCTION NewControlDefProc(userRoutine: ControlDefProcPtr): ControlDefUPP;
FUNCTION NewUserItemProc(userRoutine: UserItemProcPtr): UserItemUPP;
FUNCTION NewListDefProc(userRoutine: ListDefProcPtr): ListDefUPP;
FUNCTION NewTEClickLoopProc(userRoutine: TEClickLoopProcPtr): TEClickLoopUPP;
FUNCTION NewAEEventHandlerProc(userRoutine: AEEventHandlerProcPtr): AEEventHandlerUPP;
FUNCTION NewModalFilterProc(userRoutine: ModalFilterProcPtr): ModalFilterUPP;
FUNCTION NewListSearchProc(userRoutine: ListSearchProcPtr): ListSearchUPP;
FUNCTION NewDeviceLoopDrawingProc(userRoutine: DeviceLoopDrawingProcPtr):
                                        DeviceLoopDrawingUPP;
FUNCTION NewVBLProc(userRoutine: VBLProcPtr): VBLUPP;
FUNCTION NewSoundProc(userRoutine: SoundProcPtr): SoundUPP;
NewControlKeyFilterProc                 (userRoutine)
FUNCTION NewControlUserPaneDrawProc(userRoutine: ControlUserPaneDrawProcPtr):
                                        ControlUserPaneDrawUPP;
FUNCTION NewControlUserPaneHitTestProc(userRoutine: ControlUserPaneHitTestProcPtr):
                                        ControlUserPaneHitTestUPP;
FUNCTION NewControlUserPaneTrackingProc(userRoutine: ControlUserPaneTrackingProcPtr):
                                        ControlUserPaneTrackingUPP;
FUNCTION NewControlUserPaneIdleProc(userRoutine: ControlUserPaneIdleProcPtr):
                                        ControlUserPaneIdleUPP;
FUNCTION NewControlUserPaneKeyDownProc(userRoutine: ControlUserPaneKeyDownProcPtr):
                                        ControlUserPaneKeyDownUPP;
FUNCTION NewControlUserPaneActivateProc(userRoutine: ControlUserPaneActivateProcPtr):
                                        ControlUserPaneActivateUPP;
FUNCTION NewControlUserPaneFocusProc(userRoutine: ControlUserPaneFocusProcPtr):
                                        ControlUserPaneFocusUPP;
FUNCTION NewControlUserPaneBackgroundProc(userRoutine: ControlUserPaneBackgroundProcPtr):
                                        ControlUserPaneBackgroundUPP;
```