This chapter describes the Mixed Mode Manager, the part of the Macintosh system software that manages the mixed-mode architecture of PowerPC processor-based computers running 680x0-based code (including system software, applications, and stand-alone code modules). The Mixed Mode Manager cooperates with the 68LC040 Emulator to provide a fast, efficient, and virtually transparent method for code in one instruction set architecture to call code in another architecture. The Mixed Mode Manager handles all the details of switching between architectures.

The Mixed Mode Manager is intended to operate transparently to most applications and other software. You need the information in this chapter only if

- you want to recompile your application into PowerPC code and your application passes the address of some routine to the system software using a reference of type `ProcPtr`
- your application—written in either PowerPC or 680x0 code—supports installable code modules that might be written in a different architecture
- you are writing stand-alone code (for example, a VBL task or a component) that could be called from either the PowerPC native environment or the 680x0 emulated environment
- you are writing a debugger or other software that needs to know about the structure of the stack at any time (for example, during a mode switch)

You do not need to read this chapter if you're simply writing 680x0 code that doesn't call external code modules of unknown type, or if you are writing PowerPC code that calls other PowerPC code using a procedure pointer. In these cases, any environment switching that might occur is handled completely transparently by the Mixed Mode Manager.

IMPORTANT
This chapter describes the operation and features of the Mixed Mode Manager and the 68LC040 Emulator as they exist in the first version of the system software for PowerPC processor-based Macintosh computers. ▲

To use this chapter, you should already be generally familiar with the Macintosh Operating System. See the books *Inside Macintosh: Processes* and *Inside Macintosh: Memory* for information about the run-time architecture of the 680x0 environment. You also need to be familiar with the run-time architecture of PowerPC processor-based Macintosh computers, as explained in the chapter "Introduction to PowerPC System Software."

This chapter begins by describing the mixed-mode architecture of PowerPC processor-based Macintosh computers and the operations of the Mixed Mode Manager. Then it shows how to use the Mixed Mode Manager to call external code.

About the Mixed Mode Manager

The Mixed Mode Manager is the part of the Macintosh Operating System that allows PowerPC processor-based Macintosh computers to cooperatively run 680x0 applications, PowerPC applications, 680x0 system software, and PowerPC system software. It provides a number of capabilities, including

- transparent access to 680x0-based system software from PowerPC applications
- transparent access to PowerPC processor-based system software from 680x0 applications
- a method—independent of the instruction set architecture—of calling an external piece of code. This includes
  - transparent access to PowerPC code by 680x0 applications
  - system support for calling 680x0 code from PowerPC code
  - system support for calling PowerPC code from 680x0 code
- support for patching PowerPC or 680x0 code with PowerPC or 680x0 code
- support for stand-alone code resources containing either 680x0 or PowerPC code

In short, the Mixed Mode Manager is intended to provide both PowerPC processor-based and 680x0-based code transparent access to code written in another instruction set (or in an instruction set whose type is unknown). It does this by keeping track of what kind of code is currently executing and, when necessary, switching modes. For example, if some PowerPC code calls a Macintosh Operating System routine that exists only in 680x0 form, the Mixed Mode Manager translates the routine's parameters from their PowerPC arrangement (for example, stored in registers GPR3 and GPR4) into the appropriate 680x0 arrangement (for example, stored in registers D0 and D1, with the result placed into register A0).

The Mixed Mode Manager is an integral part of the system software for PowerPC processor-based Macintosh computers. It is designed to hide, as much as possible, the dual nature of the operating environment supported on PowerPC processor-based Macintosh computers running the 68LC040 Emulator. Except in specific cases described later, your application or other software should not need to call the routines provided by the Mixed Mode Manager.

External Code

To appreciate when and why you might need to use the routines provided by the Mixed Mode Manager, you need to understand the circumstances in which you might directly or indirectly call code in an instruction set architecture different from that of the calling code. There are several ways to execute **external code** (code that is not directly contained in your application or software), including

- calling a trap
- calling a device driver (for example, by calling the driver's Open, Status, or Control routines)

- loading and then executing code contained in a resource
- using the address of a procedure or function obtained from an unknown source

In any of these four cases, the external code that you call might be in an instruction set architecture that is different from the instruction set architecture of the calling code. (For example, an application that uses the PowerPC instruction set might call a ROM-based Toolbox trap that uses the 680x0 instruction set.) As a result, in all these cases, the Mixed Mode Manager might have to switch environments to allow the called routine to execute and then switch back to allow your application or other software to continue execution.

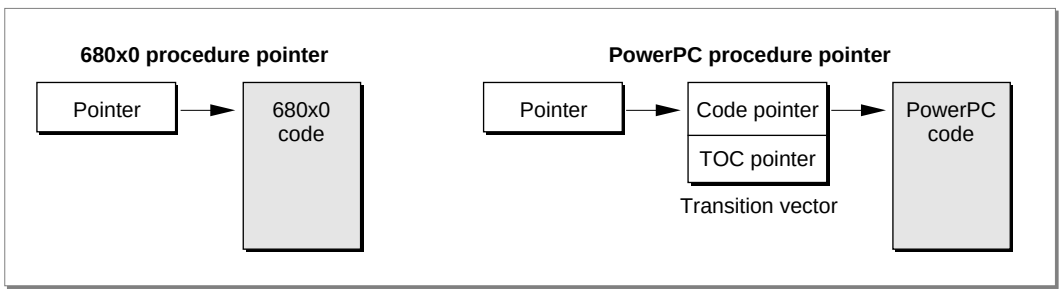
In the first two of the four cases, the Mixed Mode Manager is able to handle all required mode switching virtually transparently to the calling software. In the two last cases, however, you might need to intervene in the otherwise automatic operations of the Mixed Mode Manager. This is because the Mixed Mode Manager cannot tell, from a given pointer to some executable code, what kind of code the pointer references.

The following section describes in greater detail the extent of this problem and the way you need to solve it, using universal procedure pointers in place of procedure pointers. See "Using the Mixed Mode Manager" beginning on page 2-14 for code samples that illustrate how to create and use universal procedure pointers.

Procedure Pointers

For present purposes, a **procedure pointer** is any reference generated by a compiler when taking the address of a routine. On 680x0-based Macintosh computers, a procedure pointer is simply the address of the routine's executable code (and is defined by the `ProcPtr` data type). On PowerPC processor-based Macintosh computers, a procedure pointer is the address of the routine's transition vector. Figure 2-1 illustrates the structure of procedure pointers in each environment.

Figure 2-1 680x0 and PowerPC procedure pointers



A **transition vector** is a set of two addresses: the address of the routine's executable code and the address of the fragment's table of contents (TOC).

The Macintosh programming interfaces allow you to use procedure pointers in several ways. A procedure pointer can be

- passed as a parameter to a system software routine (for example, the `growZone` parameter to the `SetGrowZone` routine)
- passed in a field of a parameter block or other data structure (for example, the `gzProc` field of a `Zone` parameter block)
- stored in an application-specific global data structure (for example, the addresses stored in a `grafProcs` field of a graphics port)
- installed into a vector accessed through system global variables (for example, the `jGNEFilter` global variable)
- installed into the trap dispatch table or into a patch daisy chain using the `SetToolTrapAddress` or `SetOSTrapAddress` routine

As indicated previously, the Mixed Mode Manager cannot tell, from a given procedure pointer, what kind of code the pointer references (either directly through a pointer of type `ProcPtr` or indirectly through a transition vector). The Mixed Mode Manager solves this problem by requiring you to use generalized procedure pointers, known as universal procedure pointers, whenever you would previously have used a procedure pointer. A **universal procedure pointer** is either a normal 680x0 procedure pointer (that is, the address of a routine) or the address of a **routine descriptor,** a data structure that the Mixed Mode Manager uses to encapsulate information about an externally referenced routine. A routine descriptor describes the address of the routine, its parameters, and its calling conventions.

```
typedef RoutineDescriptor *UniversalProcPtr;
```

Note

See "Routine Descriptors" on page 2-37 for a description of the fields of a routine descriptor. ◆

The Macintosh application programming interfaces have been revised for the PowerPC platform to change all references to procedure pointers to references to universal procedure pointers. (The new interfaces are called the universal interface files.) For example, the `SetGrowZone` function was previously declared in the interface file `Memory.h` like this:

```
typedef ProcPtr GrowZoneProcPtr;
pascal void SetGrowZone (GrowZoneProcPtr growZone);
```

In the updated interface file `Memory.h`, `SetGrowZone` is declared like this:

```
typedef UniversalProcPtr GrowZoneUPP;
extern pascal void SetGrowZone (GrowZoneUPP growZone);
```

This redefinition of all procedure pointers as universal procedure pointers ensures that at the time a procedure is to be executed, the Operating System has enough information to determine the routine's instruction set architecture and hence to determine whether

a mode switch is necessary. In addition, if a mode switch is necessary, the universal procedure pointer (if it is a pointer to a routine descriptor) provides information about the routine's calling conventions, the number and sizes of its parameters, and so forth.

It's important to understand exactly when you need to be concerned about routine descriptors and when you need to use the new programming interfaces when writing your application. The following cases cover most of the relevant possibilities:

- If your application uses the 680x0 instruction set (and therefore executes under the 68LC040 Emulator on PowerPC processor-based Macintosh computers) and does not support external code modules, you do not need to use routine descriptors or the new programming interfaces.
- If your application uses the PowerPC instruction set, you must use the new programming interfaces.
- If your application uses either the 680x0 instruction set or the PowerPC instruction set and makes calls only to code of the same type, you do not need to create routine descriptors.
- If your code uses the PowerPC instruction set and passes a routine's address to code that might be in the 680x0 instruction set, then you need instead to pass the address of a routine descriptor. This applies to all the methods of passing a routine address listed earlier in this section (as a parameter to a system software routine, in a field of a parameter block, and so forth).
- If you create a resource containing PowerPC code that might be called either by 680x0 code or by PowerPC code, that code must be preceded by a routine descriptor. It's possible that the calling code simply loads the resource and jumps to its beginning; if the resource does not begin with a routine descriptor, the Mixed Mode Manager will not be called to determine whether a mode switch is necessary. See "Executing Resource-Based Code" on page 2-24 for more details.

IMPORTANT

In short, you need to convert procedure pointers to universal procedure pointers only if you pass a routine's address to code that is external to your application. See "Using Universal Procedure Pointers" beginning on page 2-21 for details on making the appropriate modifications to your application. ▲

Mode Switches

This section describes the operations of the Mixed Mode Manager in switching modes (from PowerPC native mode to 680x0 emulation mode, or vice versa). It describes the circumstances under which mode switches are performed and the mechanism that the Mixed Mode Manager uses to switch modes.

IMPORTANT

The information in this section is provided for debugging purposes only. Your application (or other code) should not rely on the details of mode switching presented here. ▲

Every mode switch occurs as a result of either an explicit or an implicit cross-mode call. An **explicit cross-mode call** occurs when the calling software itself calls the `CallUniversalProc` function and passes a universal procedure pointer of a routine that exists in an instruction set architecture other than that of the caller. An **implicit cross-mode call** occurs when the calling software executes a routine descriptor for a routine that exists in an instruction set architecture other than that of the caller.

The mixed-mode architecture of PowerPC processor-based computers running 680x0-based code gives rise to four possible situations when a piece of code calls a system software routine:

- When 680x0 code calls a system software routine that exists as 680x0 code, the routine is called directly, using the trap dispatch mechanism provided in the 68LC040 Emulator.
- When 680x0 code calls a system software routine that exists as PowerPC code, the routine is called indirectly, using the address—contained in the trap dispatch table—of a routine descriptor, which invokes a mode switch to the PowerPC environment. When the PowerPC code returns, the executing environment is switched back to the 68LC040 Emulator. See the next section, "Calling PowerPC Code From 680x0 Code," for more details.
- When PowerPC code calls a system software routine that exists as PowerPC code, the routine is called through glue in the system software import library. The glue code calls `CallUniversalProc`, which determines that the routine is PowerPC code and then calls it directly.
- When PowerPC code calls a system software routine that exists as 680x0 code, the routine is called through glue code contained in the system software import library. The glue code sets up a 680x0 universal procedure pointer (which is simply a 680x0 procedure pointer) and executes the 680x0 code by calling the `CallUniversalProc` function. See "Calling 680x0 Code From PowerPC Code" on page 2-12 for more details.

IMPORTANT

Only 680x0 code can make implicit cross-mode calls. Native PowerPC code must always make explicit cross-mode calls. The Mixed Mode Manager determines whether a mode switch is necessary. ▲

Calling PowerPC Code From 680x0 Code

This section describes how the Mixed Mode Manager switches modes from the 680x0 emulated environment to the PowerPC native environment. This usually happens when 680x0 code calls a system software routine that is implemented in the PowerPC instruction set.

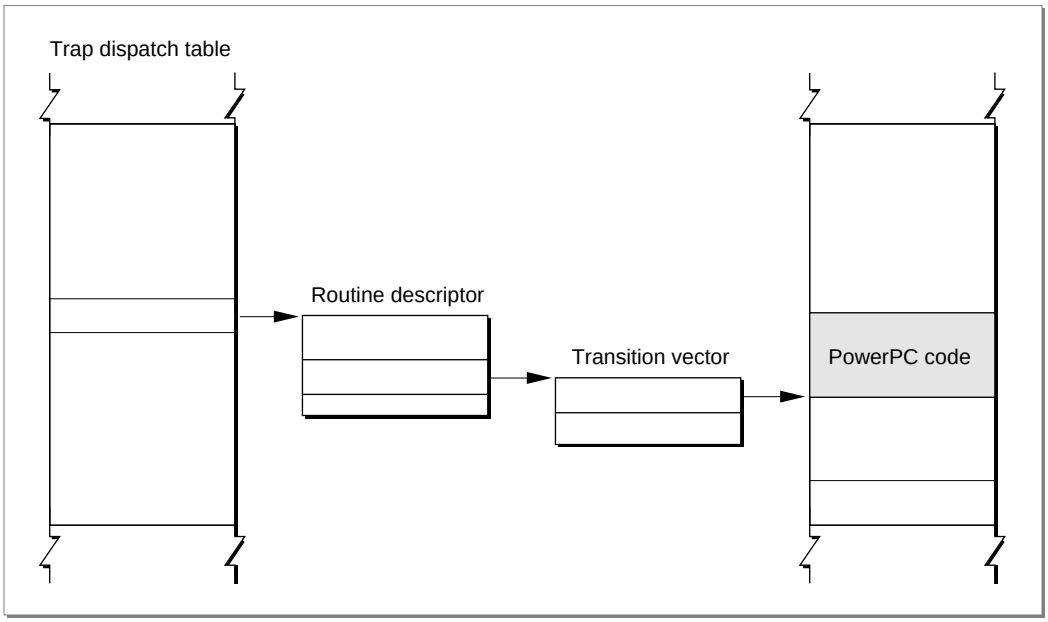
Suppose that a 680x0 application calls some system software routine. The application is not aware that it is running under the 68LC040 Emulator, so it just pushes the routine's parameters onto the stack (or stores them into registers) and then jumps to the routine or calls a trap that internally jumps to the routine. If the routine exists as 680x0 code, no mode switch is required and the routine is called as usual. If, however, the routine

exists as PowerPC code, the calling application must implicitly invoke the Mixed Mode Manager.

If the calling application merely jumps to the PowerPC code, the code must begin with a routine descriptor, as explained in "Executing Resource-Based Code" on page 2-24. If the calling application calls a trap, the trap dispatch table must contain—instead of the address of the routine's executable code—the address of a routine descriptor for that routine. This routine descriptor is created at system startup time.

Figure 2-2 shows the path followed when a 680x0 application calls a system software routine implemented as PowerPC code. The trap dispatch table contains the address of the native routine's routine descriptor. The routine descriptor contains the address of the routine's transition vector, which in turn contains the routine's entry point and TOC value.

Figure 2-2 Calling PowerPC code from a 680x0 application

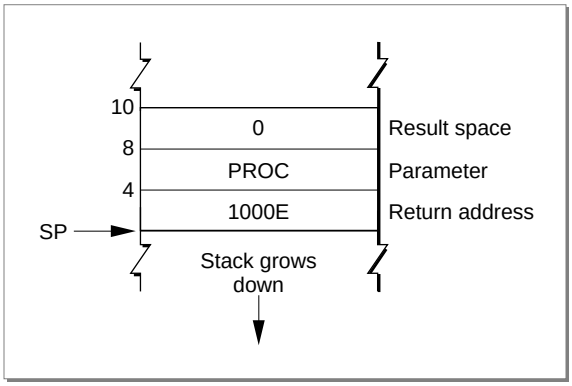


For example, suppose that your application calls the `CountResources` function, as follows:

```
myResCount = CountResources('PROC');
```

Suppose further that `CountResources` has been ported to the PowerPC instruction set. When your application calls `CountResources`, the stack looks like the one shown in Figure 2-3.

Figure 2-3 The stack before a mode switch

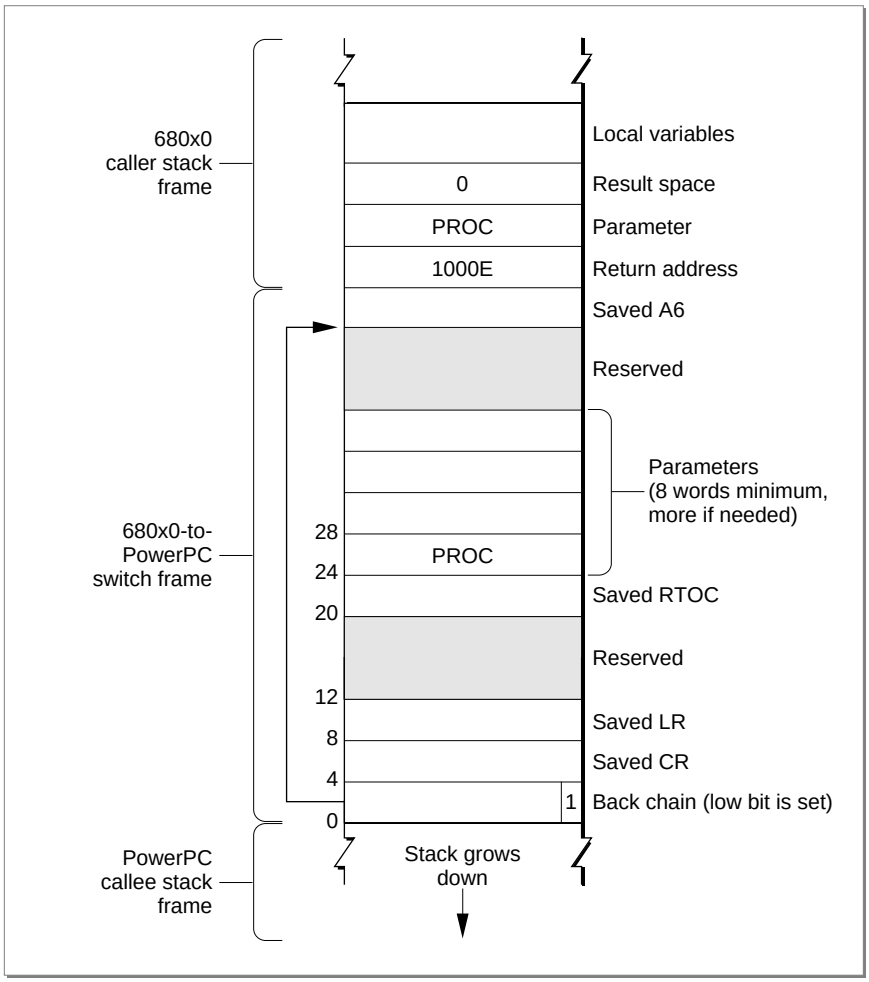


The trap dispatcher executes the `CountResources` routine descriptor, which begins with an executable instruction that invokes the Mixed Mode Manager. The Mixed Mode Manager retrieves the transition vector and creates a switch frame on the stack. A **switch frame** is a stack frame that contains information about the routine to be executed, the state of various registers, and the address of the previous frame. Figure 2-4 shows the structure of a 680x0-to-PowerPC switch frame.

IMPORTANT

Notice in Figure 2-4 that the low-order bit in the back chain pointer to the saved A6 value is set. The Mixed Mode Manager uses that bit internally as a signal that a switch frame is on the stack. The Mixed Mode Manager will fail if the stack pointer has an odd value. ▲

Figure 2-4 A 680x0-to-PowerPC switch frame



In addition to creating a switch frame, the Mixed Mode Manager also sets up several CPU registers:

- The Table of Contents Register (RTOC) must be set to the TOC address of the fragment containing the `CountResources` routine. This value is obtained from the transition vector whose address is extracted from the routine descriptor.
- The Link Register (LR) must be set to point to code that cleans up the stack and restarts the emulator.

At this point, it's safe to execute the native `CountResources` code. When `CountResources` completes, the Mixed Mode Manager copies the return value from R3 into its proper location (in a register or on the stack). The RTOC, LR, and CR are restored to their saved values, and the switch frame is popped off the stack. The Mixed Mode Manager also pops the return address off the stack, as well as the parameters of routines

of type `pascal`. Finally, the Mixed Mode Manager jumps back into the 68LC040 Emulator and the application continues execution.

Calling 680x0 Code From PowerPC Code

This section describes how the Mixed Mode Manager switches modes from the PowerPC native environment to the 680x0 emulated environment. This usually happens when PowerPC code calls a system software routine that is implemented in the 680x0 instruction set.

For example, suppose that a PowerPC application calls a system software routine that exists only as 680x0 code. In the system software import library must exist a small piece of glue code that

- allocates space on the stack for the routine's result, if any
- determines the address of the 680x0 routine from the trap dispatch table
- provides the procedure information for the routine
- calls the `CallUniversalProc` function

Listing 2-1 illustrates a sample glue routine for the QuickDraw text-measuring routine `TextWidth`.

IMPORTANT
Glue routines like the one illustrated in Listing 2-1 are part of the system software import library. You do not need to write glue routines like this. ▲

Listing 2-1 Sample glue code for a 680x0 routine

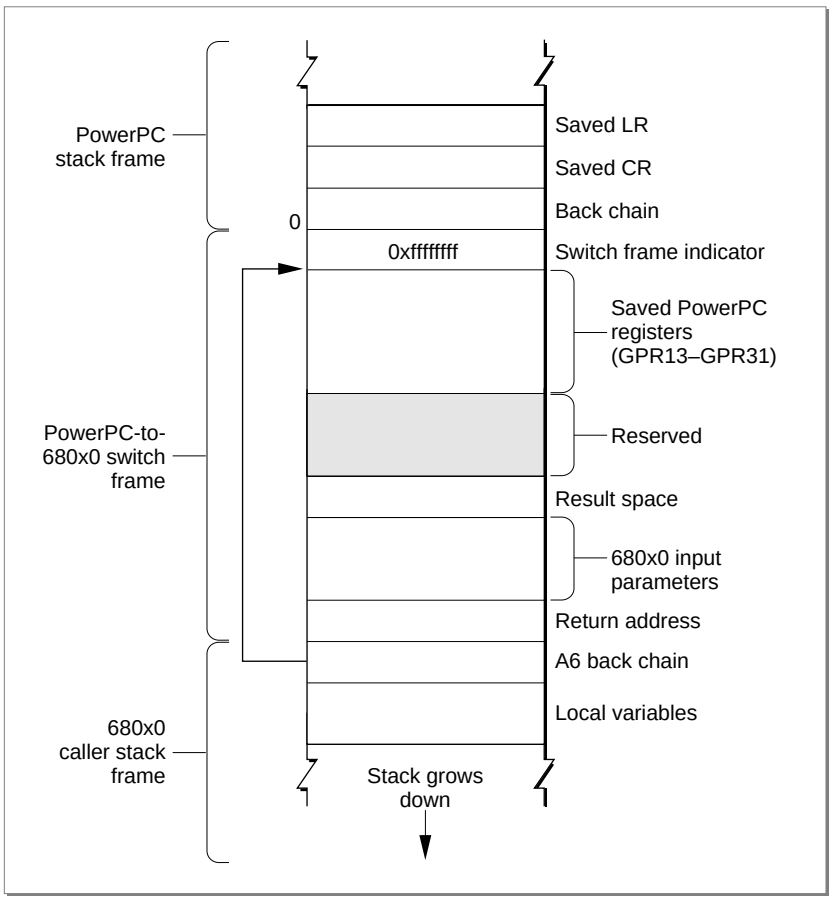
```
enum {
   uppTextWidthProcInfo =  kPascalStackBased
                     | RESULT_SIZE(kTwoByteCode)
                     | STACK_ROUTINE_PARAMETER(1, kFourByteCode)
                     | STACK_ROUTINE_PARAMETER(2, kTwoByteCode)
                     | STACK_ROUTINE_PARAMETER(3, kTwoByteCode)
};

short TextWidth (Ptr textBuf, short firstByte, short byteCount)
{
   ProcPtr        textWidth_68K;

   textWidth_68K = NGetTrapAddress(_TextWidth, ToolTrap);
   return CallUniversalProc((UniversalProcPtr)textWidth_68K,
               uppTextWidthProcInfo, textBuf, firstByte, byteCount);
}
```

See "Specifying Procedure Information" beginning on page 2-14 for a description of the constants and macros used to define the procedure information (that is, the `myProcInfo` parameter).

Note
For Operating System traps (that is, traps of type `OSTrap`), the trap dispatcher copies the trap number into register D1. As a result, the glue code illustrated in Listing 2-1 would need to call the function `CallOSTrapUniversalProc`. ◆

The call to `CallUniversalProc` invokes the Mixed Mode Manager, which verifies that a mode switch is necessary. At that point, the Mixed Mode Manager saves all nonvolatile registers and other necessary information on the stack in a switch frame. Figure 2-5 shows the structure of a PowerPC-to-680x0 switch frame.

Figure 2-5 A PowerPC-to-680x0 switch frame



Once the switch frame is set up, the Mixed Mode Manager sets up the 68LC040 Emulator's context block and then jumps into the emulator. When the routine has finished executing, it attempts to jump to the return address pushed onto the stack. That return address points to a mode-switching structure contained in the Reserved area in the switch frame. The emulator encounters the instruction in the `goMixedModeTrap` field of the routine descriptor and then saves the current 680x0 state in its context block. Once this is done, the Mixed Mode Manager restores native registers that were previously saved and deallocates the switch frame. Control then returns to the caller of `CallUniversalProc`.

IMPORTANT

As currently implemented, the instruction that causes a return from the 68LC040 Emulator to the native PowerPC environment clears the low-order 5 bits of the Condition Code Register (CCR). This prevents 680x0 callback procedures from returning information in the CCR. If you want to port 680x0 code that calls an external routine that returns results in the CCR, you must instead call a 680x0 stub that saves that information in some other place. ▲

Using the Mixed Mode Manager

You can use the Mixed Mode Manager to specify the procedure information for a routine, create routine descriptors, and execute the code referenced by a universal procedure pointer. Typically, you'll call `NewRoutineDescriptor` to create a routine descriptor and `CallUniversalProc` to execute the code described by a routine descriptor. You can dispose of routine descriptors you no longer need by calling the `DisposeRoutineDescriptor` function.

Remember that if you are compiling code for the 680x0 environment, you don't need to worry about creating, calling, or disposing of routine descriptors. For 680x0 code, the compiler variable `USESROUTINEDESCRIPTORS` is set to `false` (the default setting). Any calls in your source code to the `NewRoutineDescriptor` function are replaced by the code address passed as a parameter to `NewRoutineDescriptor`. Similarly, any calls to `DisposeRoutineDescriptor` are simply removed.

Note

Your development environment sets the `USESROUTINEDESCRIPTOR` variable to the value appropriate for the kind of code you are compiling, You don't need to set or reset this variable. ◆

Specifying Procedure Information

The primary task of the Mixed Mode Manager is to convert routine parameters between the 680x0 and PowerPC environments. The parameter passing conventions in the PowerPC environment are identical for all routines, so you'll need to specify the calling conventions only for 680x0 routines.

In the Macintosh Operating System, there are five basic kinds of calling conventions:

- Pascal routines with the parameters passed on the stack
- C routines with the parameters passed on the stack
- routines with the parameters passed in registers
- dispatched Pascal or C routines with the selector in a register and the parameters on the stack
- dispatched Pascal routines with the selector and the parameters on the stack

In addition to these five basic kinds of calling conventions, there exist a number of cases that the Mixed Mode Manager treats specially. For example, an ADB service routine is passed information in registers A0, A1, A2, and D0.

The Mixed Mode Manager uses a long word of type `ProcInfoType` to encode a routine's **procedure information,** which contains essential information about the calling conventions and other features of a routine. You need to specify procedure information when you create a new routine descriptor by calling the `NewRoutineDescriptor` function.

```
typedef unsigned long ProcInfoType;
```

IMPORTANT

In all likelihood, you do not need to read the remainder of this section, which explains in detail the structure of the `ProcInfoType` long word and shows how to create custom procedure information. The universal interface files define procedure information for each universal procedure pointer used by the system. For example, the interfaces define the constant `uppGrowZoneProcInfo` for you to use when specifying the procedure information for a grow-zone function. You need to create procedure information only for routines not defined in the programming interfaces. You can probably skip to the section "Using Universal Procedure Pointers" on page 2-21. ▲

The lower-order 4 bits of the procedure information encode the routine's calling conventions. You specify calling conventions using these constants:

```
enum {
   /*calling conventions*/
   kPascalStackBased                   = (CallingConventionType)0,
   kCStackBased                        = (CallingConventionType)1,
   kRegisterBased                      = (CallingConventionType)2,
   kThinkCStackBased                   = (CallingConventionType)5,
   kD0DispatchedPascalStackBased       = (CallingConventionType)8,
   kD0DispatchedCStackBased            = (CallingConventionType)9,
   kD1DispatchedPascalStackBased       = (CallingConventionType)12,
   kStackDispatchedPascalStackBased    = (CallingConventionType)14,
   kSpecialCase                        = (CallingConventionType)15
};
```

For example, a routine that passes its parameters on the stack according to normal C language conventions would have the rightmost 4 bits of the procedure information set to 0001 (hexadecimal 0x00000001).

Except for routines having calling conventions of type `kSpecialCase`, the 2 bits to the left of the calling convention bits encode the size of the result returned by the routine. You can access those bits using a constant:

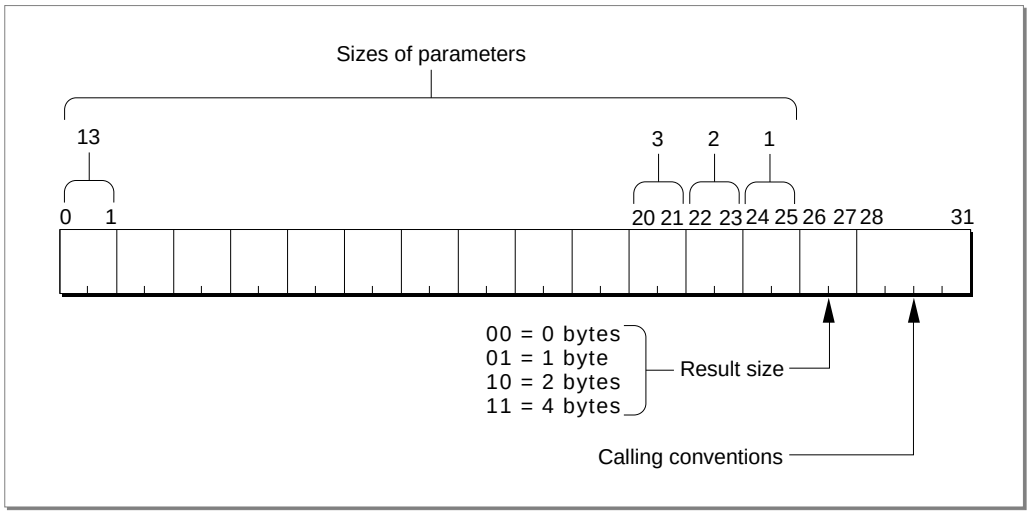
```
#define kResultSizePhase            4
```

The Mixed Mode Manager provides four constants and a macro that you can use to set a routine's result size in its procedure information.

```
enum {
   kNoByteCode                  = 0,
   kOneByteCode                 = 1,
   kTwoByteCode                 = 2,
   kFourByteCode                = 3
};

#define RESULT_SIZE(sizeCode) \
              ((ProcInfoType)(sizeCode) << kResultSizePhase)
```

Except as already noted, every set of procedure information uses its rightmost 6 bits to specify the calling conventions and result size information. The calling conventions, which take up the rightmost 4 bits, determine how the remaining bits of a routine's procedure information are interpreted. For example, if the rightmost 4 bits contain the value `kCStackBased` or the value `kPascalStackBased`, then the remaining bits encode the sizes and number of the parameters passed on the stack. Figure 2-6 shows how the Mixed Mode Manager interprets the procedure information for a stack-based routine.

Figure 2-6 Procedure information for a stack-based routine



Once again, the Mixed Mode Manager provides a set of constants and macros that you can use to specify a stack-based routine's procedure information.

```
#define kStackParameterPhase 6
#define kStackParameterWidth 2

#define STACK_ROUTINE_PARAMETER(whichParam, sizeCode) \
   ((ProcInfoType)(sizeCode) << (kStackParameterPhase + \
                  (((whichParam) - 1) * kStackParameterWidth)))
```

As you can see, the maximum number of stack-based parameters whose sizes you can specify using a variable of type `ProcInfoType` is 13. The procedure information encoding used by the Mixed Mode Manager places limits on the number of specifiable register-based parameters as well. See Table 3-1 at the end of this section (page 2-20) for a complete list of these limits.

The new application programming interface files described earlier (on page 2-6) include constants that define procedure information for each type of routine to which you might need to create a universal procedure pointer. For example, the interface file `Memory.h` includes these definitions:

```
enum {
   uppGrowZoneProcInfo = kPascalStackBased
      | RESULT_SIZE(SIZE_CODE(sizeof(long)))
      | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Size))),
   uppPurgeProcProcInfo = kPascalStackBased
      | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
};
```

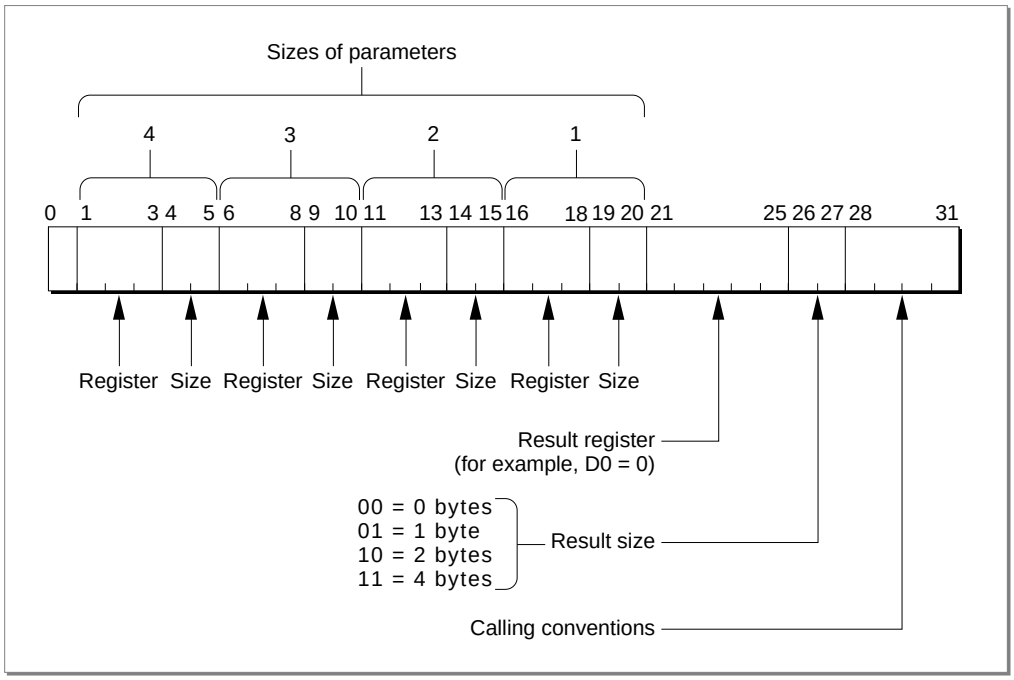
A grow-zone function follows normal Pascal calling conventions, returns a value that is 4 bytes long, and takes a single 4-byte parameter on the stack. A purge-warning procedure follows normal Pascal calling conventions, returns no value, and takes a single 4-byte parameter on the stack.

The Mixed Mode Manager provides similar constants and macros for specifying procedure information for register-based routines.

```
#define kRegisterResultLocationPhase                    \
                           (kCallingConventionWidth + kResultSizeWidth)
#define kRegisterResultLocationWidth                    5
#define kRegisterParameterPhase                         \
                           (kCallingConventionWidth + kResultSizeWidth + \
                              kRegisterResultLocationWidth)
#define kRegisterParameterWidth                         5
#define kRegisterParameterWhichPhase                    2
#define kRegisterParameterSizePhase                     0
#define kDispatchedSelectorSizeWidth                    2

#define kDispatchedSelectorSizePhase                    \
                           (kCallingConventionWidth + kResultSizeWidth)
#define kDispatchedParameterPhase                       \
                           (kCallingConventionWidth + kResultSizeWidth + \
                              kDispatchedSelectorSizeWidth)
#define REGISTER_RESULT_LOCATION(whichReg) \
   ((ProcInfoType)(whichReg) << kRegisterResultLocationPhase)
#define REGISTER_ROUTINE_PARAMETER(whichParam, whichReg, sizeCode) \
   ((((ProcInfoType)(sizeCode) << kRegisterParameterSizePhase) | \
   ((ProcInfoType)(whichReg) << kRegisterParameterWhichPhase)) << \
   (kRegisterParameterPhase + (((whichParam)- 1) * kRegisterParameterWidth)))
```

For example, Figure 2-7 shows the arrangement of the procedure information for a register-based routine.

Figure 2-7 Procedure information for a register-based routine



The register fields use the following constants to encode 680x0 register information:

```
enum {
   /*680x0 registers*/
   kRegisterD0                  = 0,
   kRegisterD1                  = 1,
   kRegisterD2                  = 2,
   kRegisterD3                  = 3,
   kRegisterD4                  = 8,
   kRegisterD5                  = 9,
   kRegisterD6                  = 10,
   kRegisterD7                  = 11,
   kRegisterA0                  = 4,
   kRegisterA1                  = 5,
   kRegisterA2                  = 6,
   kRegisterA3                  = 7,
   kRegisterA4                  = 12,
   kRegisterA5                  = 13,
   kRegisterA6                  = 14,
   kCCRegisterCBit              = 16,
   kCCRegisterVBit              = 17,
   kCCRegisterZBit              = 18,
```

```
   kCCRegisterNBit              = 19,
   kCCRegisterXBit              = 20
};
```

Note
The result size should be specified as 0 for results returned in any of the CCR registers. ◆

The Mixed Mode Manager also provides constants and macros to specify the procedure information for stack-based routines that take a register-based selector and for stack-based routines that take a stack-based selector.

Note
See "Procedure Information" beginning on page 2-27 for a complete description of the constants you can use to specify a routine's procedure information. See "C Language Macros for Defining Procedure Information" on page 2-50 for a complete list of the Mixed Mode Manager macros you can use to create procedure information. ◆

As noted earlier, there are limits on the number of parameters that a procedure information can describe. Table 3-1 lists the available calling conventions and the maximum number of specifiable parameters and selectors for each convention.

IMPORTANT
The input parameters can be passed in any of the registers D0–D3 and A0–A3; the output parameter can be returned in any register. ▲

Table 3-1 Limits on the number of specifiable parameters in a procedure information

Calling convention	Maximum number of parameters	Number of selectors
`kPascalStackBased`	13	0
`kCStackBased`	13	0
`kRegisterBased`	4 input, 1 output	0
`kThinkCStackBased`	13	0
`kD0DispatchedPascalStackBased`	12	1
`kD0DispatchedCStackBased`	12	1
`kD1DispatchedPascalStackBased`	12	1
`kStackDispatchedPascalStackBased`	12	1

In general, these limitations should not affect you. There are, however, a very few cases in which the documented behavior of a routine prevents it from being implemented in native PowerPC code. For example, the low-level .ENET driver routines `ReadRest` and `ReadPacket` return information in several registers. As a result, they cannot be implemented natively. (Because these routines are typically called only in code where

speed of execution is critical, it's not likely that you would want to incur the overhead of a mode switch by writing native callbacks to the .ENET driver.)

Using Universal Procedure Pointers

When you call the `NewRoutineDescriptor` or `NewFatRoutineDescriptor` function to create a routine descriptor, the Mixed Mode Manager calls the Memory Manager to allocate a nonrelocatable block in the current heap in which to store the new routine descriptor. Eventually, you might want to dispose of the space occupied by the routine descriptor; you can do this by calling the `DisposeRoutineDescriptor` function.

In general, there are two ways you'll probably handle this allocation and deallocation. By far the easiest method is to allocate in your application's heap, at application initialization time, a routine descriptor for each routine whose address you'll need to pass elsewhere. For example, if your application calls `TrackControl` with a custom action procedure, you can create a routine descriptor in the application heap when your application starts up, as shown in Listing 2-2.

Listing 2-2 Creating global routine descriptors

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

Later you would call `TrackControl` like this:

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

The routine descriptor pointed to by the global variable `myActionProc` remains allocated until your application quits, at which time the Process Manager reclaims all the memory in your application heap.

Note
If you don't want `TrackControl` to call an application-defined action procedure, you must pass `NULL` in place of `myActionProc`. In that case, you don't need to call `NewRoutineDescriptor`. ◆

The other way to handle routine descriptors is to create them as you need them and then dispose of them as soon as you're finished with them. This practice would be useful for routines you don't call very often. Listing 2-3 shows a way to call the `ModalDialog` function to display a rarely used modal dialog box.

Listing 2-3 Creating local routine descriptors

```
void DoAboutBox (void)
{
   short                myItem = 0;
   DialogPtr            myDialog;
   UniversalProcPtr     myModalProc;

   myDialog = GetNewDialog(kAboutBoxID, NULL, (WindowPtr) -1L);
   myModalProc = NewRoutineDescriptor((ProcPtr)MyEventFilter,
                                      uppModalFilterProcInfo,
                                      GetCurrentISA());
   while (myItem != iOK)
      ModalDialog(myModalProc, &myItem);
   DisposeDialog(myDialog);
   DisposeRoutineDescriptor(myModalProc);
}
```

If you decide to allocate and dispose of routine descriptors locally, make sure that you don't dispose of a routine descriptor before it's actually used by the Operating System. (This could happen, for instance, if you pass a universal procedure pointer for a completion routine and then exit the local procedure before the completion routine is called.)

Note
You should call `DisposeRoutineDescriptor` only to dispose routine descriptors that you created using either `NewRoutineDescriptor` or `NewFatRoutineDescriptor`. ◆

Using Static Routine Descriptors

Instead of allocating space for routine descriptors in your application heap (as described in the previous section), you can also create routine descriptors on the stack or in your global variable space by using macros supplied by the Mixed Mode Manager. Most likely, you'll create a descriptor on the stack when you need to use a routine descriptor for a very short time. For example, you could use the function defined in Listing 2-4 instead of the one defined in Listing 2-3.

Listing 2-4 Creating static routine descriptors

```
void DoAboutBox (void)
{
   short                myItem = 0;
   DialogPtr            myDialog;
   RoutineDescriptor    myRD =
                           BUILD_ROUTINE_DESCRIPTOR(uppModalFilterProcInfo,
                                                    (ProcPtr)MyEventFilter);
   UniversalProcPtr     myModalProc;

   myDialog = GetNewDialog(kAboutBoxID, NULL, (WindowPtr) -1L);
   myModalProc = @myRD;
   while (myItem != iOK)
      ModalDialog(myModalProc, &myItem);
   DisposeDialog(myDialog);
}
```

As you can see, the `DoAboutBox` function defined in Listing 2-4 uses the macro `BUILD_ROUTINE_DESCRIPTOR` to create a routine descriptor on the stack and then passes the address of that routine descriptor to the `ModalDialog` procedure. Because the routine descriptor is created on the stack, there is no need to dispose of it before exiting the `DoAboutBox` function.

You can create a routine descriptor in your application's global data area by using the `BUILD_ROUTINE_DESCRIPTOR` macro as follows:

```
static RoutineDescriptor   myRD =
                           BUILD_ROUTINE_DESCRIPTOR(uppModalFilterProcInfo,
                                                    (ProcPtr)MyEventFilter);
```

This line of code creates a routine descriptor as part of the application global variables. The advantage of this method is that you don't have to call `NewRoutineDescriptor` to allocate a routine descriptor in your heap.

The C language macro `BUILD_ROUTINE_DESCRIPTOR` is defined in Listing 2-5.

Listing 2-5 Building a static routine descriptor

```
#define BUILD_ROUTINE_DESCRIPTOR(procInfo, procedure)                  \
{                                                                      \
   _MixedModeMagic,                    /*mixed-mode A-trap*/           \
   kRoutineDescriptorVersion,          /*version*/                     \
   kSelectorsAreNotIndexable,          /*RD flags: not dispatched*/    \
   0,                                  /*reserved1*/                   \
   0,                                  /*reserved2*/                   \
```

```
   0,                                  /*selector info*/               \
   0,                                  /*number of routines*/          \
   {                                   /*it's an array*/               \
      {                                /*it's a structure*/            \
         (procInfo),                   /*the procedure info*/          \
         0,                            /*reserved*/                    \
         kPowerPCISA,                  /*ISA*/                         \
         kProcDescriptorIsAbsolute |   /*flags: absolute address*/      \
         kFragmentIsPrepared |         /*it's prepared*/               \
         kUseNativeISA,                /*always use native ISA*/       \
         (ProcPtr)(procedure),         /*the procedure*/               \
         0,                            /*reserved*/                    \
         0,                            /*not dispatched*/              \
      },                                                               \
   },                                                                  \
}
```

IMPORTANT

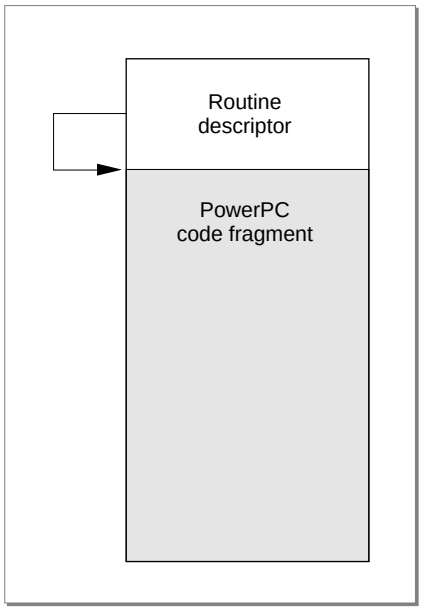
You should use the `BUILD_ROUTINE_DESCRIPTOR` macro only to create a routine descriptor that describes a nondispatched routine that exists as PowerPC code. ▲

The Mixed Mode Manager also defines a C language macro that you can use to create static fat routine descriptors. See the Mixed Mode Manager interface file for the definition of the `BUILD_FAT_ROUTINE_DESCRIPTOR` macro.

Executing Resource-Based Code

As you've seen earlier in this book (in the section "Executable Resources" on page 1-34), you can create executable resources that contain PowerPC code to serve as accelerated versions of 680x0 code resources. The accelerated resource is simply a PowerPC version of the 680x0 code resource, prefixed with a routine descriptor for the code contained in the resource. The routine descriptor is necessary for the Mixed Mode Manager to know whether it needs to change modes in order to execute the code. The routine descriptor also lets the Mixed Mode Manager know whether it needs to call the Code Fragment Manager to prepare the fragment. Figure 2-8 shows the structure your code-containing resources should have.

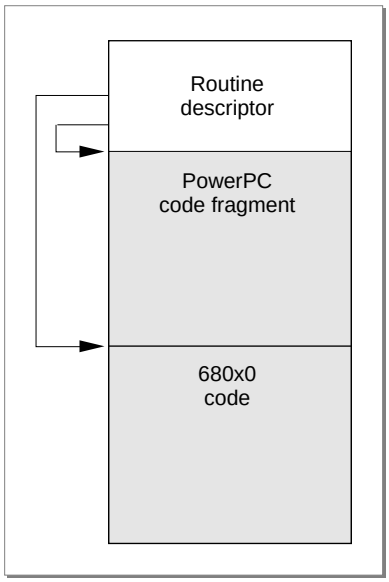
Figure 2-8 General structure of an executable code resource



The `procDescriptor` field of the routine record—contained in the `routineRecords` field of the routine descriptor—should contain the offset from the beginning of the resource (that is, the beginning of the routine descriptor) to the beginning of the executable code fragment. In addition, the routine flags for the specified code should have the `kProcDescriptorIsRelative` bit set, indicating that the address is relative, not absolute. If the code contained in the resource is PowerPC code, you should also set the `kFragmentNeedsPreparing` bit.

It's also possible to create "fat" code-bearing resources, that is, resources containing both 680x0 and PowerPC versions of some routine. Figure 2-9 shows the general structure of such a resource.

Figure 2-9 General structure of a fat resource



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. As with any code-bearing resource, the `procDescriptor` field of each routine record should contain the offset from the beginning of the resource to the beginning of the appropriate code. The flags for both routine records should have the `kProcDescriptorIsRelative` flag set, and the routine flags for the PowerPC routine record should have the `kFragmentNeedsPreparing` flag set.

The MPW interface file `MixedMode.r` provides Rez templates that you can use to create the accelerated resource shown in Figure 2-8 or the fat resource shown in Figure 2-9.

▲ **WARNING**
Do not call accelerated resources at interrupt time unless you are certain that the resource has already been loaded into memory, locked, and prepared for execution. If the resource containing the code hasn't been prepared, the Code Fragment Manager will attempt to do so, and thereby allocate memory. (Memory allocation is not allowed at interrupt time.) ▲

Mixed Mode Manager Reference

This section describes the constants, data structures, and routines provided by the Mixed Mode Manager. See "Using the Mixed Mode Manager" beginning on page 2-14 for detailed instructions on using these routines.

Constants

This section describes the constants provided by the Mixed Mode Manager. You use these constants to specify routine descriptor flags and a routine's procedure information. Because the universal interface files define procedure information for the most common callback routines, it's likely that you won't need to use the procedure information constants listed here.

Routine Descriptor Flags

The `routineDescriptorFlags` field of a routine descriptor contains a set of routine descriptor flags that specify attributes of the described routine. You can use constants to specify the routine descriptor flags. In general, you should use the constant `kSelectorsAreNotIndexable` when constructing your own routine descriptors; the value `kSelectorsAreIndexable` is reserved for use by Apple.

```
enum {
   kSelectorsAreNotIndexable     = (RDFlagsType)0x00,
   kSelectorsAreIndexable        = (RDFlagsType)0x01
};
```

Constant descriptions

`kSelectorsAreNotIndexable`
: For dispatched routines, the recognized routine selectors are not contiguous.

`kSelectorsAreIndexable`
: For dispatched routines, the recognized routine selectors are contiguous and therefore indexable.

Procedure Information

The Mixed Mode Manager uses a long word of type `ProcInfoType` to encode a routine's procedure information, which contains essential information about the calling conventions and other features of a routine. These values specify

- the routine's calling conventions
- the sizes and locations of the routine's parameters, if any
- the size and location of the routine's result, if any

See "Specifying Procedure Information" beginning on page 2-14 for a description of the general structure of a routine's procedure information. The Mixed Mode Manager provides a number of constants that you can use to specify the procedure information.

The following constants are used to specify the size (in bytes) of a value encoded in a routine's procedure information.

```
enum {
   /*size codes*/
   kNoByteCode                   = 0,
   kOneByteCode                  = 1,
   kTwoByteCode                  = 2,
   kFourByteCode                 = 3
};
```

Constant descriptions

`kNoByteCode` The value occupies no bytes.

`kOneByteCode` The value occupies 1 byte.

`kTwoByteCode` The value occupies 2 bytes.

`kFourByteCode` The value occupies 4 bytes.

The offsets to fields and the widths of the fields within a value of type `ProcInfoType` are defined by constants:

```
/*offsets to and widths of procedure information fields*/
#define kCallingConventionPhase                     0
#define kCallingConventionWidth                     4
#define kResultSizePhase                            kCallingConventionWidth
#define kResultSizeWidth                            2
#define kResultSizeMask                             0x30
#define kStackParameterPhase                        6
#define kStackParameterWidth                        2
#define kRegisterResultLocationPhase                \
                          (kCallingConventionWidth + kResultSizeWidth)
#define kRegisterResultLocationWidth                5
#define kRegisterParameterPhase                     \
                          (kCallingConventionWidth + kResultSizeWidth + \
                             kRegisterResultLocationWidth)
#define kRegisterParameterWidth                     5
#define kRegisterParameterWhichPhase                2
#define kRegisterParameterSizePhase                 0
#define kDispatchedSelectorSizeWidth                2
#define kDispatchedSelectorSizePhase                \
                          (kCallingConventionWidth + kResultSizeWidth)
#define kDispatchedParameterPhase                   \
                          (kCallingConventionWidth + kResultSizeWidth + \
                             kDispatchedSelectorSizeWidth)
```

Constant descriptions

`kCallingConventionPhase`
The offset from the least significant bit in the procedure information to the calling convention information.

`kCallingConventionWidth`
: The number of bits in the procedure information that encode the calling convention information.

`kResultSizePhase`
: The offset from the least significant bit in the procedure information to the function result size information.

`kResultSizeWidth`
: The number of bits in the procedure information that encode the function result size information.

`kResultSizeMask`
: A mask for the bits in the procedure information that encode the function result size information.

`kStackParameterPhase`
: The offset from the least significant bit in the procedure information to the stack parameter information.

`kStackParameterWidth`
: The number of bits in the procedure information that encode the size of a stack-based parameter.

`kRegisterResultLocationPhase`
: The offset from the least significant bit in the procedure information to the result register information.

`kRegisterResultLocationWidth`
: The number of bits in the procedure information that encode which register the result will be stored in.

`kRegisterParameterPhase`
: The offset from the least significant bit in the procedure information to the register parameter information.

`kRegisterParameterWidth`
: The number of bits in the procedure information that encode the information about a register-based parameter.

`kRegisterParameterWhichPhase`
: The offset from the beginning of a register parameter information field to the encoded register.

`kRegisterParameterSizePhase`
: The offset from the beginning of a register parameter information field to the encoded size of the parameter.

`kDispatchedSelectorSizeWidth`
: The number of bits in the procedure information that encode the size of a routine-dispatching selector.

`kDispatchedSelectorSizePhase`
: The offset from the least significant bit in the procedure information to the selector size information of a routine that is dispatched though a selector.

`kDispatchedParameterPhase`
: The offset from the least significant bit in the procedure information to the parameter information of a routine that is dispatched though a selector.

The following constants are used to specify a routine's calling conventions:

```
enum {
   /*calling conventions*/
   kPascalStackBased                   = (CallingConventionType)0,
   kCStackBased                        = (CallingConventionType)1,
   kRegisterBased                      = (CallingConventionType)2,
   kThinkCStackBased                   = (CallingConventionType)5,
   kD0DispatchedPascalStackBased       = (CallingConventionType)8,
   kD0DispatchedCStackBased            = (CallingConventionType)9,
   kD1DispatchedPascalStackBased       = (CallingConventionType)12,
   kStackDispatchedPascalStackBased    = (CallingConventionType)14,
   kSpecialCase                        = (CallingConventionType)15
};
```

Constant descriptions

`kPascalStackBased`
The routine follows normal Pascal calling conventions.

`kCStackBased` The routine follows the C calling conventions employed by the MPW development environment.

`kRegisterBased`
The parameters are passed in registers.

`kThinkCStackBased`
The routine follows the C calling conventions employed by the THINK C software development environment. Arguments are passed on the stack from right to left, and a result is returned in register D0. All arguments occupy an even number of bytes on the stack. An argument having the size of a `char` is passed in the high-order byte. You should always provide function prototypes; failure to do so may cause THINK C to generate code that is incompatible with this parameter-passing convention.

`kD0DispatchedPascalStackBased`
The parameters are passed on the stack according to Pascal conventions, and the routine selector is passed in register D0.

`kD0DispatchedCStackBased`
The parameters are passed on the stack according to C conventions, and the routine selector is passed in register D0.

`kD1DispatchedPascalStackBased`
The parameters are passed on the stack according to Pascal conventions, and the routine selector is passed in register D1.

`kStackDispatchedPascalStackBased`
The routine selector and the parameters are passed on the stack.

`kSpecialCase` The routine is a special case. You can use the following constants to specify a special case.

```
enum {
   /*special cases*/
   kSpecialCaseHighHook          = 0,
   kSpecialCaseCaretHook         = kSpecialCaseHighHook,
   kSpecialCaseEOLHook           = 1,
   kSpecialCaseWidthHook         = 2,
   kSpecialCaseNWidthHook        = 3,
   kSpecialCaseTextWidthHook     = kSpecialCaseWidthHook,
   kSpecialCaseDrawHook          = 4,
   kSpecialCaseHitTestHook       = 5,
   kSpecialCaseTEFindWord        = 6,
   kSpecialCaseProtocolHandler   = 7,
   kSpecialCaseSocketListener    = 8,
   kSpecialCaseTERecalc          = 9,
   kSpecialCaseTEDoText          = 10,
   kSpecialCaseGNEFilterProc     = 11,
   kSpecialCaseMBarHook          = 12
};
```

Constant descriptions

`kSpecialCaseHighHook`
The routine follows the calling conventions documented in *Inside Macintosh: Text;* a rectangle is on the stack and a pointer is in register A3; no result is returned.

`kSpecialCaseCaretHook`
The routine follows the calling conventions documented in *Inside Macintosh: Text;* a rectangle is on the stack and a pointer is in register A3; no result is returned.

`kSpecialCaseEOLHook`
Parameters are passed to the routine in registers A3, A4, and D0, and output is returned in the Z flag of the Status Register. An `EOLHook` routine has these calling conventions.

`kSpecialCaseWidthHook`
Parameters are passed to the routine in registers A0, A3, A4, D0, and D1, and output is returned in register D1. A `WIDTHHook` routine has these calling conventions.

`kSpecialCaseNWidthHook`
Parameters are passed to the routine in registers A0, A2, A3, A4, D0, and D1, and output is returned in register D1. An `nWIDTHHook` routine has these calling conventions.

`kSpecialCaseTextWidthHook`
Parameters are passed to the routine in registers A0, A3, A4, D0, and D1, and output is returned in register D1. A `TextWidthHook` routine has these calling conventions.

`kSpecialCaseDrawHook`
Parameters are passed to the routine in registers A0, A3, A4, D0, and

D1, and no result is returned. A `DRAWHook` routine has these calling conventions.

`kSpecialCaseHitTestHook`
Parameters are passed to the routine in registers A0, A3, A4, D0, D1, and D2, and output is returned in registers D0, D1, and D2. A `HITTESTHook` routine has these calling conventions.

`kSpecialCaseTEFindWord`
Parameters are passed to the routine in registers A3, A4, D0, and D2, and output is returned in registers D0 and D1. A `TEFindWord` hook has these calling conventions.

`kSpecialCaseProtocolHandler`
Parameters are passed to the routine in registers A0, A1, A2, A3, A4, and in the low-order word of register D1; output is returned in the Z flag of the Status Register. A protocol handler has these calling conventions.

`kSpecialCaseSocketListener`
Parameters are passed to the routine in registers A0, A1, A2, A3, A4, in the low-order byte of register D0, and in the low-order word of register D1; output is returned in the Z flag of the Status Register. A socket listener has these calling conventions.

`kSpecialCaseTERecalc`
Parameters are passed to the routine in registers A3 and D7, and output is returned in registers D2, D3, and D4. A TextEdit line-start recalculation routine has these calling conventions.

`kSpecialCaseTEDoText`
Parameters are passed to the routine in registers A3, D3, D4, and D7, and output is returned in registers A0 and D0. A TextEdit text-display, hit-test, and caret-positioning routine has these calling conventions.

`kSpecialCaseGNEFilterProc`
Parameters are passed to the routine in registers A1 and D0 and on the stack, and output is returned on the stack. A `GetNextEvent` filter procedure has these calling conventions.

`kSpecialCaseMBarHook`
Parameters are passed to the routine on the stack, and output is returned in register D0. A menu bar hook routine has these calling conventions.

For register-based routines, the registers are encoded in the routine's procedure information using these constants:

```
enum {
   /*680x0 registers*/
   kRegisterD0                  = 0,
   kRegisterD1                  = 1,
   kRegisterD2                  = 2,
   kRegisterD3                  = 3,
   kRegisterD4                  = 8,
```

```
   kRegisterD5                  = 9,
   kRegisterD6                  = 10,
   kRegisterD7                  = 11,
   kRegisterA0                  = 4,
   kRegisterA1                  = 5,
   kRegisterA2                  = 6,
   kRegisterA3                  = 7,
   kRegisterA4                  = 12,
   kRegisterA5                  = 13,
   kRegisterA6                  = 14,
   kCCRegisterCBit              = 16,
   kCCRegisterVBit              = 17,
   kCCRegisterZBit              = 18,
   kCCRegisterNBit              = 19,
   kCCRegisterXBit              = 20
};
```

Constant descriptions

`kRegisterD0` Register D0.

`kRegisterD1` Register D1.

`kRegisterD2` Register D2.

`kRegisterD3` Register D3.

`kRegisterD4` Register D4.

`kRegisterD5` Register D5.

`kRegisterD6` Register D6.

`kRegisterD7` Register D7.

`kRegisterA0` Register A0.

`kRegisterA1` Register A1.

`kRegisterA2` Register A2.

`kRegisterA3` Register A3.

`kRegisterA4` Register A4.

`kRegisterA5` Register A5.

`kRegisterA6` Register A6.

`kCCRegisterCBit`
The C (carry) flag of the Status Register.

`kCCRegisterVBit`
The V (overflow) flag of the Status Register.

`kCCRegisterZBit`
The Z (zero) flag of the Status Register.

`kCCRegisterNBit`
The N (negative) flag of the Status Register.

`kCCRegisterXBit`
The X (extend) flag of the Status Register.

Routine Flags

The `routineFlags` field of a routine record contains a set of flags that specify information about a routine. You can use constants to specify the desired routine flags. Currently, only 5 of the 16 bits in a routine flags word are defined. You should set all the other bits to 0.

```
enum {
   kProcDescriptorIsAbsolute     = (RoutineFlagsType)0x00,
   kProcDescriptorIsRelative     = (RoutineFlagsType)0x01
};
```

Constant descriptions

`kProcDescriptorIsAbsolute`
The address of the routine's entry point specified in the `procDescriptor` field of a routine record is an absolute address.

`kProcDescriptorIsRelative`
The address of the routine's entry point specified in the `procDescriptor` field of a routine record is relative to the beginning of the routine descriptor. If the code is contained in a resource and its absolute location is not known until run time, you should set this flag.

```
enum {
   kFragmentIsPrepared           = (RoutineFlagsType)0x00,
   kFragmentNeedsPreparing       = (RoutineFlagsType)0x02
};
```

Constant descriptions

`kFragmentIsPrepared`
The fragment containing the code to be executed is already loaded into memory and prepared by the Code Fragment Manager.

`kFragmentNeedsPreparing`
The fragment containing the code to be executed needs to be loaded into memory and prepared by the Code Fragment Manager. If this flag is set, the `kPowerPCISA` and `kProcDescriptorIsRelative` flags should also be set.

```
enum {
   kUseCurrentISA                = (RoutineFlagsType)0x00,
   kUseNativeISA                 = (RoutineFlagsType)0x04
};
```

Constant descriptions

`kUseCurrentISA` If possible, use the current instruction set architecture when executing a routine.

`kUseNativeISA` Use the native instruction set architecture when executing a routine.

```
enum {
   kPassSelector                = (RoutineFlagsType)0x00,
   kDontPassSelector            = (RoutineFlagsType)0x08
};
```

Constant descriptions

`kPassSelector` Pass the routine selector to the target routine as a parameter.

`kDontPassSelector`
Do not pass the routine selector to the target routine as a parameter. You should not use this flag for 680x0 routines.

```
enum {
   kRoutineIsNotDispatchedDefaultRoutine
                                = (RoutineFlagsType)0x00,
   kRoutineIsDispatchedDefaultRoutine
                                = (RoutineFlagsType)0x10
};
```

Constant descriptions

`kRoutineIsNotDispatchedDefaultRoutine`
This routine is not the default routine for a set of routines that is dispatched using a routine selector.

`kRoutineIsDispatchedDefaultRoutine`
This routine is the default routine for a set of routines that is dispatched using a routine selector. If a set of routines is dispatched using a routine selector and the routine corresponding to a specified selector cannot be found, this default routine is called. This routine must be able to accept the same procedure information for all routines. If possible, it is passed the procedure information passed in a call to `CallUniversalProc`.

IMPORTANT

In general, you should use the constants `kPassSelector` and `kRoutineIsNotDispatchedDefaultRoutine`. The constants `kDontPassSelector` and `kRoutineIsDispatchedDefaultRoutine` are reserved for use with selector-based system software routines. ▲

Instruction Set Architectures

The `ISA` field of a routine record contains a flag that specifies the instruction set architecture of a routine. You can use constants to specify the instruction set architecture.

```
enum {
   kM68kISA                    = (ISAType)0,     /*MC680x0 architecture*/
   kPowerPCISA                 = (ISAType)1      /*PowerPC architecture*/
};
```

Constant descriptions

`kM68kISA`	The routine consists of 680x0 code.
`kPowerPCISA`	The routine consists of PowerPC code.

Data Structures

This section describes the two data structures provided by the Mixed Mode Manager:

- the routine record, which contains information about a routine's calling conventions, the sizes and locations of its parameters, and its location in memory
- the routine descriptor, which provides a generalization of procedure pointers (variables of type `ProcPtr`) common in the 680x0 environment

Routine Records

A routine record is a data structure that contains information about a particular routine. The routine descriptor specifies, among other things, the instruction set architecture of the routine, the number and size of the routine's parameters, the routine's calling conventions, and the routine's location in memory. At least one routine record is contained in the `routineRecords` field of a routine descriptor. A routine record is defined by the `RoutineRecord` data type.

```
struct RoutineRecord {
   ProcInfoType         procInfo;         /*calling conventions*/
   unsigned char        reserved1;        /*reserved*/
   ISAType              ISA;              /*instruction set architecture*/
   RoutineFlagsType     routineFlags;     /*flags for each routine*/
   ProcPtr              procDescriptor;   /*the thing we're calling*/
   unsigned long        reserved2;        /*reserved*/
   unsigned long        selector;         /*selector for dispatched calls*/
};
typedef struct RoutineRecord RoutineRecord;
typedef RoutineRecord *RoutineRecordPtr, **RoutineRecordHandle;
```

Field descriptions

`procInfo`	A value of type `ProcInfoType` that encodes essential information about the routine's calling conventions and parameters. See "Procedure Information" beginning on page 2-27 for a complete list of the constants you can use to set this field.
`reserved1`	Reserved. This field must be 0.

`ISA`	The instruction set architecture of the routine. See "Instruction Set Architectures" beginning on page 2-35 for a complete listing of the constants you can use to set this field.
`routineFlags`	A value of type `RoutineFlagsType` that contains a set of flags describing the routine. See "Routine Flags" beginning on page 2-34 for a complete listing of the constants you can use to set this field.
`procDescriptor`	A pointer to the routine's code. If the routine consists of 680x0 code and the `kProcDescriptorIsAbsolute` flag is set in the `routineFlags` field, then this field contains the address of the routine's entry point. If the routine consists of 680x0 code and the `kProcDescriptorIsRelative` flag is set, then this field contains the offset from the beginning of the routine descriptor to the routine's entry point. If the routine consists of PowerPC code, the `kFragmentIsPrepared` flag is set, and the `kProcDescriptorIsAbsolute` flag is set, then this field contains the address of the routine's transition vector. If the routine consists of PowerPC code, the `kFragmentNeedsPreparing` flag is set, and the `kProcDescriptorIsRelative` flag is set, then this field contains the offset from the beginning of the routine descriptor to the routine's entry point.
`reserved2`	Reserved. This field must be 0.
`selector`	Reserved. This field must be 0. For routines that are dispatched, this field contains the routine selector.

Routine Descriptors

A routine descriptor is a data structure used by the Mixed Mode Manager to execute a routine. The external interface to a routine descriptor is through a universal procedure pointer, of type `UniversalProcPtr`, which is defined as a procedure pointer (if the code is 680x0 code) or as a pointer to a routine descriptor (if the code is PowerPC code). A routine descriptor is defined by the `RoutineDescriptor` data type.

```
struct RoutineDescriptor {
   unsigned short       goMixedModeTrap;  /*mixed-mode A-trap*/
   char                 version;          /*routine descriptor version*/
   RDFlagsType          routineDescriptorFlags;
                                          /*routine descriptor flags*/
   unsigned long        reserved1;        /*reserved*/
   unsigned char        reserved2;        /*reserved*/
   unsigned char        selectorInfo;     /*selector information*/
   short                routineCount;     /*index of last RR in this RD*/
   RoutineRecord        routineRecords[1];/*the individual routines*/
};
typedef struct RoutineDescriptor RoutineDescriptor;
```

Field descriptions

`goMixedModeTrap`
: An A-line instruction that is used privately by the Mixed Mode Manager. When the emulator encounters this instruction, it transfers control to the Mixed Mode Manager. This field contains the value $AAFE.

`version`
: The version number of the `RoutineDescriptor` data type. The current version number is defined by the constant `kRoutineDescriptorVersion`:

```
enum {kRoutineDescriptorVersion = 7};
```

`routineDescriptorFlags`
: A set of routine descriptor flags. Currently, all the bits in this field should be set to 0, unless you are specifying a routine descriptor for a dispatched routine. See "Routine Descriptor Flags" on page 2-27 for a complete description of these flags.

`reserved1`
: Reserved. This field must initially be 0.

`reserved2`
: Reserved. This field must be 0.

`selectorInfo`
: Reserved. This field must be 0.

`routineCount`
: The index of the final routine record in the following array, `routineRecords`. Because the `routineRecords` array is zero-based, this field does not contain an actual count of the routine records contained in that array. Often, you'll use a routine descriptor to describe a single procedure, in which case this field should contain the value 0. You can, however, construct a routine descriptor that contains pointers to both 680x0 and PowerPC code (known as a "fat" routine descriptor). In that case, this field should contain the value 1.

`routineRecords`
: An array of routine records for the routines described by this routine descriptor. See "Routine Records" on page 2-36 for the structure of a routine record. This array is zero-based.

IMPORTANT

Your application (or other software) should never attempt to guide its execution by inspecting the value in the `ISA` field of a routine record and jumping to the address in the `procDescriptor` field. ▲

Mixed Mode Manager Routines

This section describes the routines provided by the Mixed Mode Manager. You can use these routines to

- create and dispose of routine descriptors
- execute routines described by routine descriptors

In general, you need to call these routines only from PowerPC code. To maintain a single source code base for your software, however, you can call Mixed Mode Manager routines from 680x0 code, as long as you set the `USESROUTINEDESCRIPTORS` compiler flag to `false` (its default setting). To compile code for the PowerPC environment, you should set the `USESROUTINEDESCRIPTORS` flag to `true`.

See "Using the Mixed Mode Manager" beginning on page 2-14 for detailed instructions on using these routines.

Creating and Disposing of Routine Descriptors

The Mixed Mode Manager provides routines that you can use to create and dispose of routine descriptors. In general, you need to create routine descriptors only for routines whose addresses are exported to the system software (for example, a completion procedure). You don't need to create a routine descriptor for a routine that is called by code of the same type.

NewRoutineDescriptor

You can call the `NewRoutineDescriptor` function to create a new routine descriptor.

```
pascal UniversalProcPtr NewRoutineDescriptor
                    (ProcPtr theProc, ProcInfoType theProcInfo,
                       ISAType theISA);
```

`theProc` The address of the routine.

`theProcInfo`
The procedure information to be associated with the routine.

`theISA` The instruction set architecture of the routine being described.

DESCRIPTION

The `NewRoutineDescriptor` function creates a new routine descriptor and returns a pointer (of type `UniversalProcPtr`) to it. If the value of the `theProc` parameter is `NULL`, `NewRoutineDescriptor` returns the value `NULL`.

The memory occupied by the new routine descriptor is allocated in the current heap. If you want the memory to be allocated in some other heap, you'll need to set the current heap to that heap and then restore the current heap before exiting.

SPECIAL CONSIDERATIONS

The `NewRoutineDescriptor` function allocates memory; you should not call it at interrupt time or from any code that might be executed when memory is low. In addition, the block of memory allocated by `NewRoutineDescriptor` is nonrelocatable.

To help minimize heap fragmentation, you should try to allocate any routine descriptors you will need early in your application's execution.

When the `USESROUTINEDESCRIPTORS` compile flag is `false`, the `NewRoutineDescriptor` function simply returns the address passed in the `theProc` parameter and does not allocate memory for a routine descriptor.

SEE ALSO

See "Using Universal Procedure Pointers" beginning on page 2-21 for a more complete description of when and how to create routine descriptors. See "Specifying Procedure Information" beginning on page 2-14 for information on creating procedure information.

NewFatRoutineDescriptor

You can call the `NewFatRoutineDescriptor` function to create a new fat routine descriptor.

```
pascal UniversalProcPtr NewFatRoutineDescriptor
                    (ProcPtr theM68kProc, ProcPtr thePowerPCProc,
                       ProcInfoType theProcInfo);
```

`theM68kProc`
The address of a 680x0 routine.

`thePowerPCProc`
The address of a PowerPC routine.

`theProcInfo`
The procedure information to be associated with the routine.

DESCRIPTION

The `NewFatRoutineDescriptor` function creates a new fat routine descriptor and returns a pointer (of type `UniversalProcPtr`) to it. The routine descriptor contains routine records for both 680x0 and PowerPC versions of a routine. If the value of either the `theM68kProc` parameter or the `thePowerPCProc` parameter is `NULL`, `NewFatRoutineDescriptor` returns the value `NULL`.

The memory occupied by the new routine descriptor is allocated in the current heap. If you want the memory to be allocated in some other heap, you'll need to set the current heap to that heap and then restore the original heap before exiting.

SPECIAL CONSIDERATIONS

The `NewFatRoutineDescriptor` function allocates memory; you should not call it at interrupt time or from any code that might be executed when memory is low. In addition, the block of memory allocated by `NewFatRoutineDescriptor` is nonrelocatable. To

help minimize heap fragmentation, you should try to allocate any routine descriptors you will need early in your application's execution.

When the `USESROUTINEDESCRIPTORS` compile flag is `false`, the `NewFatRoutineDescriptor` function is undefined.

SEE ALSO

See "Using Universal Procedure Pointers" beginning on page 2-21 for a more complete description of when and how to create routine descriptors. See "Specifying Procedure Information" beginning on page 2-14 for information on creating procedure information.

DisposeRoutineDescriptor

You can call the `DisposeRoutineDescriptor` function to dispose of a routine descriptor.

```
pascal void DisposeRoutineDescriptor
                                    (UniversalProcPtr theProcPtr);
```

`theProcPtr`
A universal procedure pointer.

DESCRIPTION

The `DisposeRoutineDescriptor` function disposes of the routine descriptor pointed to by the `theProcPtr` parameter. You should call this function to release any memory allocated by a previous call to `NewRoutineDescriptor`.

The Operating System automatically disposes of any remaining routine descriptors held by your application when `ExitToShell` is executed on its behalf. As a result, you don't need to explicitly dispose of any routine descriptors that you have allocated in your application heap.

SPECIAL CONSIDERATIONS

Be careful not to dispose of a routine descriptor that is still in use by the Operating System. Code that installs completion routines or other routines called asynchronously may complete before the completion routine is actually called.

When the `USESROUTINEDESCRIPTORS` compile flag is `false`, the `DisposeRoutineDescriptor` function does nothing.

Calling Routines via Universal Procedure Pointers

The Mixed Mode Manager provides a function that allows you to execute the routine associated with a universal procedure pointer. It also provides a function that allows you to call the routine associated with a universal procedure pointer, following Operating System register saving and restoring conventions.

CallUniversalProc

You can use the `CallUniversalProc` function to call the routine associated with a universal procedure pointer.

```
long CallUniversalProc (UniversalProcPtr theProcPtr,
                        ProcInfoType theProcInfo, ...);
```

`theProcPtr`
A universal procedure pointer.

`theProcInfo`
The procedure information associated with the routine specified by the `theProcPtr` parameter.

DESCRIPTION

The `CallUniversalProc` function executes the routine associated with the specified universal procedure pointer. You pass `CallUniversalProc` a universal procedure pointer (which may be either a 680x0 procedure pointer or the address of the routine descriptor), a set of procedure information, and a variable number of parameters that are passed to the routine. `CallUniversalProc` returns a result of type `long` that contains the result (if any) returned by the called routine.

SPECIAL CONSIDERATIONS

If the universal procedure pointer passed to `CallUniversalProc` is the address of the routine descriptor, that routine descriptor must already exist before you call `CallUniversalProc`. If you pass the address of an invalid routine descriptor to `CallUniversalProc`, a system error will occur.

CallOSTrapUniversalProc

You can call the `CallOSTrapUniversalProc` function to call the routine associated with a universal procedure pointer, following Operating System register saving and

restoring conventions. You're likely to need to use this function only if you need to patch an Operating System trap.

```
long CallOSTrapUniversalProc (UniversalProcPtr theProcPtr,
                              ProcInfoType theProcInfo, ...);
```

`theProcPtr`
A universal procedure pointer.

`theProcInfo`
The procedure information associated with the routine specified by the `theProcPtr` parameter.

DESCRIPTION

The `CallOSTrapUniversalProc` function executes the routine associated with the specified universal procedure pointer, following standard conventions for executing Operating System traps. Registers A1, A2, D1, and D2 are saved before the routine is executed and restored after its completion; in addition, register A0 is saved and restored, depending on the setting of the appropriate flag bit in the trap word. The trap number is put into register D1; you should make certain to record that fact in any procedure information you build yourself.

You pass `CallOSTrapUniversalProc` a universal procedure pointer (which may be either a 680x0 procedure pointer or the address of a routine descriptor), a set of procedure information, and a variable number of parameters that are passed to the routine. `CallOSTrapUniversalProc` returns a result of type `long` that contains the result (if any) returned by the called routine.

SPECIAL CONSIDERATIONS

If the universal procedure pointer passed to `CallOSTrapUniversalProc` is the address of the routine descriptor, that routine descriptor must already exist before you call `CallOSTrapUniversalProc`. If you pass the address of an invalid routine descriptor to `CallOSTrapUniversalProc`, a system error will occur.

The `CallOSTrapUniversalProc` function is defined only for register-based Operating System traps. Make sure that the procedure information specified in the `theProcInfo` parameter correctly specifies the calling conventions of the trap. In particular, do not specify either C or Pascal calling conventions.

Determining Instruction Set Architectures

The Mixed Mode Manager contains a function that you can use to determine the current instruction set architecture.

GetCurrentISA

You can use the `GetCurrentISA` function to get the current instruction set architecture.

```
ISAType GetCurrentISA (void);
```

DESCRIPTION

The `GetCurrentISA` function returns the current instruction set architecture. See "Instruction Set Architectures" on page 2-35 for a list of the values `GetCurrentISA` can return.

SPECIAL CONSIDERATIONS

Currently, the `GetCurrentISA` function is defined as a compiler macro.

```
#if defined(powerc) || defined(__powerc)
#define GetCurrentISA()       ((ISAType) kPowerPCISA)
#else
#define GetCurrentISA()       ((ISAType) kM68kISA)
#endif
```

The implementation details are subject to change.

Summary of the Mixed Mode Manager

C Summary

Constants

```
/*Gestalt selector and response bits*/
#define gestaltMixedModeAttr     'mixd'   /*Mixed Mode Mgr attributes*/
enum {
   gestaltPowerPCAware           = 0      /*true if MMMgr supports PowerPC*/
};

enum {
   /*current version of RoutineDescriptor data type*/
   kRoutineDescriptorVersion     = 7
};
```

Routine Flags

```
enum {
   kProcDescriptorIsAbsolute     = (RoutineFlagsType)0x00,
   kProcDescriptorIsRelative     = (RoutineFlagsType)0x01
};

enum {
   kFragmentIsPrepared           = (RoutineFlagsType)0x00,
   kFragmentNeedsPreparing       = (RoutineFlagsType)0x02
};

enum {
   kUseCurrentISA                = (RoutineFlagsType)0x00,
   kUseNativeISA                 = (RoutineFlagsType)0x04
};

enum {
   kPassSelector                 = (RoutineFlagsType)0x00,
   kDontPassSelector             = (RoutineFlagsType)0x08
};
```

```
enum {
   kRoutineIsNotDispatchedDefaultRoutine
                                = (RoutineFlagsType)0x00,
   kRoutineIsDispatchedDefaultRoutine
                                = (RoutineFlagsType)0x10
};
```

Instruction Set Architectures

```
enum {
   kM68kISA                     = (ISAType)0,     /*MC680x0 architecture*/
   kPowerPCISA                  = (ISAType)1      /*PowerPC architecture*/
};
```

Routine Descriptor Flags

```
enum {
   kSelectorsAreNotIndexable    = (RDFlagsType)0x00,
   kSelectorsAreIndexable       = (RDFlagsType)0x01
};
```

Procedure Information

```
enum {
   /*size codes*/
   kNoByteCode                  = 0,
   kOneByteCode                 = 1,
   kTwoByteCode                 = 2,
   kFourByteCode                = 3
};

/*offsets to and widths of procedure information fields*/
#define kCallingConventionPhase                     0
#define kCallingConventionWidth                     4
#define kResultSizePhase                            kCallingConventionWidth
#define kResultSizeWidth                            2
#define kResultSizeMask                             0x30
#define kStackParameterPhase                        6
#define kStackParameterWidth                        2
#define kRegisterResultLocationPhase                \
                          (kCallingConventionWidth + kResultSizeWidth)
#define kRegisterResultLocationWidth                5
```

```
#define kRegisterParameterPhase                         \
                           (kCallingConventionWidth + kResultSizeWidth + \
                              kRegisterResultLocationWidth)
#define kRegisterParameterWidth                         5
#define kRegisterParameterWhichPhase                    2
#define kRegisterParameterSizePhase                     0
#define kDispatchedSelectorSizeWidth                    2
#define kDispatchedSelectorSizePhase                    \
                           (kCallingConventionWidth + kResultSizeWidth)
#define kDispatchedParameterPhase                       \
                           (kCallingConventionWidth + kResultSizeWidth + \
                              kDispatchedSelectorSizeWidth)

enum {
   /*calling conventions*/
   kPascalStackBased                  = (CallingConventionType)0,
   kCStackBased                       = (CallingConventionType)1,
   kRegisterBased                     = (CallingConventionType)2,
   kThinkCStackBased                  = (CallingConventionType)5,
   kD0DispatchedPascalStackBased      = (CallingConventionType)8,
   kD0DispatchedCStackBased           = (CallingConventionType)9,
   kD1DispatchedPascalStackBased      = (CallingConventionType)12,
   kStackDispatchedPascalStackBased   = (CallingConventionType)14,
   kSpecialCase                       = (CallingConventionType)15
};

enum {
   /*special cases*/
   kSpecialCaseHighHook          = 0,
   kSpecialCaseCaretHook         = kSpecialCaseHighHook,
   kSpecialCaseEOLHook           = 1,
   kSpecialCaseWidthHook         = 2,
   kSpecialCaseNWidthHook        = 3,
   kSpecialCaseTextWidthHook     = kSpecialCaseWidthHook,
   kSpecialCaseDrawHook          = 4,
   kSpecialCaseHitTestHook       = 5,
   kSpecialCaseTEFindWord        = 6,
   kSpecialCaseProtocolHandler   = 7,
   kSpecialCaseSocketListener    = 8,
   kSpecialCaseTERecalc          = 9,
   kSpecialCaseTEDoText          = 10,
   kSpecialCaseGNEFilterProc     = 11,
   kSpecialCaseMBarHook          = 12
};
```

```
enum {
   /*680x0 registers*/
   kRegisterD0                  = 0,
   kRegisterD1                  = 1,
   kRegisterD2                  = 2,
   kRegisterD3                  = 3,
   kRegisterD4                  = 8,
   kRegisterD5                  = 9,
   kRegisterD6                  = 10,
   kRegisterD7                  = 11,
   kRegisterA0                  = 4,
   kRegisterA1                  = 5,
   kRegisterA2                  = 6,
   kRegisterA3                  = 7,
   kRegisterA4                  = 12,
   kRegisterA5                  = 13,
   kRegisterA6                  = 14,
   kCCRegisterCBit              = 16,
   kCCRegisterVBit              = 17,
   kCCRegisterZBit              = 18,
   kCCRegisterNBit              = 19,
   kCCRegisterXBit              = 20
};
```

Data Types

```
typedef unsigned char ISAType;           /*instruction set architecture*/

typedef unsigned short CallingConventionType;   /*calling convention*/

typedef unsigned long ProcInfoType;      /*procedure information*/

typedef unsigned short RegisterSelectorType;

typedef unsigned short RoutineFlagsType;

struct RoutineRecord {
   ProcInfoType         procInfo;        /*calling conventions*/
   unsigned char        reserved1;       /*reserved*/
   ISAType              ISA;             /*instruction set architecture*/
   RoutineFlagsType     routineFlags;    /*flags for each routine*/
   ProcPtr              procDescriptor;  /*the thing we're calling*/
   unsigned long        reserved2;       /*reserved*/
   unsigned long        selector;        /*selector for dispatched calls*/
```

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

typedef unsigned char RDFlagsType;        /*routine descriptor flags*/

struct RoutineDescriptor {
   unsigned short       goMixedModeTrap;  /*mixed-mode A-trap*/
   char                 version;          /*routine descriptor version*/
   RDFlagsType          routineDescriptorFlags;
                                          /*routine descriptor flags*/
   unsigned long        reserved1;        /*reserved*/
   unsigned char        reserved2;        /*reserved*/
   unsigned char        selectorInfo;     /*selector information*/
   short                routineCount;     /*index of last RR in this RD*/
   RoutineRecord        routineRecords[1];/*the individual routines*/
};
typedef struct RoutineDescriptor RoutineDescriptor;
typedef RoutineDescriptor *UniversalProcPtr, **UniversalProcHandle;
typedef RoutineDescriptor *RoutineDescriptorPtr, **RoutineDescriptorHandle;
```


Mixed Mode Manager Routines

Creating and Disposing of Routine Descriptors

```
pascal UniversalProcPtr NewRoutineDescriptor
                            (ProcPtr theProc, ProcInfoType theProcInfo,
                             ISAType theISA);
pascal UniversalProcPtr NewFatRoutineDescriptor
                            (ProcPtr theM68kProc, ProcPtr thePowerPCProc,
                             ProcInfoType theProcInfo);
pascal void DisposeRoutineDescriptor
                            (UniversalProcPtr theProcPtr);
```

Calling Routines via Universal Procedure Pointers

```
long CallUniversalProc      (UniversalProcPtr theProcPtr,
                             ProcInfoType theProcInfo, ...);
long CallOSTrapUniversalProc
                            (UniversalProcPtr theProcPtr,
                             ProcInfoType theProcInfo, ...);
```

Determining Instruction Set Architectures

```
ISAType GetCurrentISA       (void);
```

C Language Macros for Defining Procedure Information

```
#define SIZE_CODE(size) (((size) == 4) ? kFourByteCode : \
   (((size) == 2) ? kTwoByteCode : (((size) == 1) ? kOneByteCode : 0)))

#define RESULT_SIZE(sizeCode) ((ProcInfoType)(sizeCode) << kResultSizePhase)

#define STACK_ROUTINE_PARAMETER(whichParam, sizeCode) \
   ((ProcInfoType)(sizeCode) << (kStackParameterPhase + \
                  (((whichParam) - 1) * kStackParameterWidth)))

#define DISPATCHED_STACK_ROUTINE_PARAMETER(whichParam, sizeCode) \
   ((ProcInfoType)(sizeCode) << (kDispatchedParameterPhase + \
                  (((whichParam) - 1) * kStackParameterWidth)))

#define DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(sizeCode) \
   ((ProcInfoType)(sizeCode) << kDispatchedSelectorSizePhase)

#define REGISTER_RESULT_LOCATION(whichReg) \
   ((ProcInfoType)(whichReg) << kRegisterResultLocationPhase)

#define REGISTER_ROUTINE_PARAMETER(whichParam, whichReg, sizeCode) \
   ((((ProcInfoType)(sizeCode) << kRegisterParameterSizePhase) | \
   ((ProcInfoType)(whichReg) << kRegisterParameterWhichPhase)) << \
   (kRegisterParameterPhase + (((whichParam)- 1) * kRegisterParameterWidth)))

#define SPECIAL_CASE_PROCINFO(specialCaseCode) \
   (kSpecialCase | ((ProcInfoType)(specialCaseCode) << 4))
```