Exception Manager

This chapter describes the Exception Manager, the part of the Macintosh system software that handles exceptions that occur during the execution of PowerPC applications or other software. The Exception Manager provides a simple way for your application to handle exceptions that occur in its context.

You need the information in this chapter if you need to handle exceptions that occur in native PowerPC code. If your application or other software is written in 680x0 code and therefore executes under the 68LC040 Emulator on PowerPC processor-based Macintosh computers, you do not in general need to read this chapter, because the existing 680x0 mechanism for handling exceptions is fully supported by the emulator.

IMPORTANT
The Exception Manager is available only in the system software for PowerPC processor-based Macintosh computers. In addition, not all features described here are available in the first version. For example, the Exception Manager in the first version does not return exceptions that arise during floating-point calculations. If your application performs floating-point operations and needs to handle any exceptions that arise during those operations, you should use the exception-handling mechanisms provided by the PowerPC Numerics library. See *Inside Macintosh: PowerPC Numerics* for complete information. ▲

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 with a description of exceptions and their handling in the PowerPC native environment. Then it shows how to use the Exception Manager to install your own exception handler.

About the Exception Manager

An **exception** is an error or other special condition detected by the microprocessor in the course of program execution. When an exception occurs, the Operating System transfers control synchronously to the relevant **exception handler,** which attempts to recover gracefully from the error or special condition. The kinds of errors or other conditions that give rise to exceptions differ from one processor to another. On 680x0 processors, for example, an exception is generated if the currently executing program attempts to divide by zero. By contrast, the PowerPC processor does not generate an exception under that condition.

In general, applications or other types of software (including much of the Macintosh Operating System and the Macintosh Toolbox) cannot tolerate the occurrence of exceptions. To provide some measure of protection from potentially fatal exceptions, the Operating System installs its own set of exception handlers. You can, if necessary, use the

Exception Manager to install application-specific exception handlers. Any exception handlers that you install apply only to your current context and only to exceptions that are not first intercepted and handled by the Operating System.

IMPORTANT

Not all exceptions that occur in your application's context are passed to your exception handler. Certain exceptions (for example, page faults) are handled completely by the Operating System's exception handlers. As a result, those exceptions do not affect the normal execution of your application or other software. ▲

When your exception handler is called, the Exception Manager passes it a parameter that contains information about the state of the machine at the time the exception occurred. On PowerPC processor-based Macintosh computers, this information includes

- the kind of exception that occurred
- the contents of the 32 general-purpose registers
- the contents of the special-purpose registers (such as the Link Register and the Condition Register)
- the contents of the 32 floating-point registers

Your exception handler can handle the exception in various ways. For example, it might modify the machine state and then resume execution. Similarly, your exception handler might simply transfer control to some other code. In rare instances, however, your exception handler might not be able to handle the exception; when this happens, the exception is usually fatal to your application.

Exception Contexts

In the first version of the system software for PowerPC processor-based Macintosh computers, each application can install its own exception handler, which remains the active handler as long as that application is the current application. In other words, the exception handler of the current application is called for all exceptions not intercepted by the Operating System. In general, this mechanism results in the execution of the appropriate exception handler. It's possible, however, for code you install to cause exceptions that are handled by some other application's exception handler. For instance, exceptions that arise during the asynchronous execution of code (such as VBL tasks, Time Manager tasks, and I/O completion routines) are handled by the exception handler of whatever application happens to be the current application at the time the exception occurs. If that application has not installed an exception handler, the exception might not be handled.

All asynchronous code executed in the first version of the system software for PowerPC processor-based Macintosh computers is executed under the 68LC040 Emulator, in which case the exceptions are handled using the existing 680x0 mechanisms. If, however, a routine executed asynchronously calls some code that is native PowerPC code, and if that native code causes an exception to occur, then the current application's exception handler (if any) is called to handle the exception.

Types of Exceptions

In the first version of the system software for PowerPC processor-based Macintosh computers, the following conditions can cause exceptions while your application or other software is executing in native mode:

- an attempt to write to write-protected memory
- an attempt to access (that is, read, write, or fetch) data at a logical address that is not assigned
- an attempt to execute trap instructions or other instructions that are not part of the supported application programming interface
- an attempt to execute invalid instructions or an invalid form of a valid instruction
- an attempt to execute privileged instructions when the system is not in privileged mode
- in appropriate circumstances, reaching a breakpoint
- in appropriate circumstances, reaching a trace point

The Exception Manager defines a number of **exception codes** that indicate these and other conditions. An exception code is a constant that indicates which kind of exception has occurred.

```
typedef unsigned long         ExceptionKind;    /*kind of exception*/

enum {
   /*exception codes*/
   unknownException                      = 0,     /*unknown exception type*/
   illegalInstructionException           = 1,     /*illegal instruction*/
   trapException                         = 2,     /*unknown trap type*/
   accessException                       = 3,     /*failed memory access*/
   unmappedMemoryException               = 4,     /*memory is unmapped*/
   excludedMemoryException               = 5,     /*memory is excluded*/
   readOnlyMemoryException               = 6,     /*memory is read-only*/
   unresolvablePageFaultException        = 7,     /*unresolvable page fault*/
   privilegeViolationException           = 8,     /*privilege violation*/
   traceException                        = 9,     /*trace*/
   instructionBreakpointException        = 10,    /*instruction breakpoint*/
   dataBreakpointException               = 11,    /*data breakpoint*/
   integerException                      = 12,    /*unused*/
   floatingPointException                = 13,    /*floating point*/
   stackOverflowException                = 14,    /*stack overflow*/
   terminationException                  = 15     /*task is being terminated*/
};
```

Not all of these exception codes are used in the first version of the system software for PowerPC processor-based Macintosh computers; see "Exception Kinds" on page 4-9 for a complete explanation of these constants.

Using the Exception Manager

The Exception Manager provides a routine that you can use to install an exception handler and remove an exception handler. This section describes how to use this routine and how to write an exception handler.

Installing an Exception Handler

You can install an exception handler for your application's context by calling the `InstallExceptionHandler` routine. You pass `InstallExceptionHandler` the address of your exception handler:

```
prevHandler = InstallExceptionHandler((ExceptionHandler)myHandler);
```

The `InstallExceptionHandler` function replaces any existing exception handler already installed for the current execution context (that is, for the current application) and returns the address of that previously installed handler. Listing 4-1 shows a routine that installs an exception handler as part of a wrapper around the `NewEmptyHandle` function.

Listing 4-1 Installing an exception handler

```
static jump_buf *curJmpBuf;

Handle __NewEmptyHandle (ushort trapWord)
{
   Handle              returnVal;
   OSErr               myErr;
   jmp_buf             localJump, *oldJump;
   ExceptionHandler    prevHandler;

   oldJump = curJmpBuf;                /*save current jump address*/
   curJmpBuf = &localJump;             /*install new jump address*/

   prevHandler = InstallExceptionHandler((ExceptionHandler)MyHandler);
   if (myErr = setjmp(localJump)) {
      LMSetMemErr(theErr);             /*set memory error*/
      returnVal = 0;                   /*no bytes allocated*/
   }
   else
      myErr = c_NewEmptyHandle(&returnVal, trapWord);

   InstallExceptionHandler(prevHandler);  /*restore previous handler*/
   curJmpBuf = oldJump;                   /*restore original jump address*/
   return (returnVal);
}
```

You can remove the current exception handler from your application's context by passing the value `nil` as the parameter to `InstallExceptionHandler`, as follows:

```
prevHandler = InstallExceptionHandler(nil);
```

Writing an Exception Handler

An exception handler has the following prototype:

```
typedef OSStatus (*ExceptionHandler) (ExceptionInformation *theException);
```

When your handler is called, the Exception Manager passes it the address of an **exception information record,** which contains information about the exception, such as its type and the state of the machine at the time the exception occurred. The exception information record is defined by the `ExceptionInformation` data type.

```
struct ExceptionInformation {
   ExceptionKind                    theKind;
   MachineInformation               *machineState;
   RegisterInformation              *registerImage;
   FPUInformation                   *FPUImage;
   union {
      MemoryExceptionInformation    *memoryInfo;
   } info;
};
typedef struct ExceptionInformation ExceptionInformation;
```

The `theKind` field contains an exception code. The fields `machineState` and `registerImage` contain information about the special-purpose and general-purpose registers, respectively. The values in the special-purpose registers are contained in a **machine information record,** defined by the `MachineInformation` data type.

```
struct MachineInformation {
   UnsignedWide         CTR;  /*Count Register*/
   UnsignedWide         LR;   /*Link Register*/
   UnsignedWide         PC;   /*Program Counter Register*/
   unsigned long        CR;   /*Condition Register*/
   unsigned long        XER;  /*Fixed-Point Exception Register*/
   unsigned long        MSR;  /*Machine State Register*/
};
typedef struct MachineInformation MachineInformation;
```

As you can see, this record contains primarily the values in the special-purpose registers. The values in the general-purpose registers are encoded using a structure of type `RegisterInformation`, which is effectively an array of 32 register values.

Note
For a more detailed description of the exception information record and its associated data types, see "Data Structures" beginning on page 4-12. ◆

Your exception handler can perform any actions necessary or useful for handling the exception. You might attempt to recover from the error or simply terminate your application gracefully. The specific actions you perform depend, of course, on the type of exception that has occurred. In general, however, you will probably want to use one or the other of two basic techniques for recovering from the exception.

- Your exception handler might simply transfer control away from the point of execution. For example, you might jump back into your main event loop or into some error recovery code.
- Alternatively, your exception handler might attempt to repair the cause of the exception by suitably modifying the state of the machine (as reported to your exception handler in an exception information record). You can alter any piece of that machine state, including the PC register. After you have suitably modified the relevant data, your handler should return, passing back a result code. The Exception Manager inspects the result code you return and determines what further actions to take. If you pass back `noErr`, then the Exception Manager restores the machine state to the state contained in the exception information record and resumes execution. If you pass back any other result code, the Operating System proceeds as if the exception had occurred but no exception handler was present.

Listing 4-2 shows a simple exception handler `MyHandler`.

Listing 4-2 A native exception handler

```
OSStatus MyHandler (ExceptionInformation *theException)
{
   if ((theException->theKind >= accessException)
            && (theException ->theKind <= unresolvablePageFaultException))
      longjmp(*curJmpBuf, memWZErr);
   else
      return (-1);
}
```

As you can see, the `MyHandler` exception handler looks for memory-related exceptions and, if it finds any, transfers control by calling the `longjmp` function.

▲ **WARNING**
Returning a value other than `noErr` from your exception handler is likely to cause the current application to be terminated. ▲

▲ **WARNING**
Your exception handler must be reentrant if it might itself cause any exceptions to be generated. For example, if your exception handler calls the `Debugger` or `DebugStr` routine, the trap exception (of type `trapException`) is generated. Normally, a debugger intercepts and handles those kinds of exceptions. If, however, no debugger is installed in the system, your exception handler might be called repeatedly. Eventually, the stack will grow to the lowest memory address, overwriting essential data and causing a system crash. ▲

Exception Manager Reference

This section describes the constants, data structures, and routine provided by the Exception Manager. See "Using the Exception Manager" beginning on page 4-6 for detailed instructions on using that routine.

Constants

This section describes the constants provided by the Exception Manager.

Exception Kinds

The Exception Manager indicates to your exception handler the kind of exception that has occurred by passing it an exception code. The exception kind is indicated by a constant.

Note
Some kinds of exceptions occur only on specific types of processors or only in specific system software versions. ◆

```
enum {
   /*exception codes*/
   unknownException                    = 0,     /*unknown exception type*/
   illegalInstructionException         = 1,     /*illegal instruction*/
   trapException                       = 2,     /*unknown trap type*/
   accessException                     = 3,     /*failed memory access*/
   unmappedMemoryException             = 4,     /*memory is unmapped*/
   excludedMemoryException             = 5,     /*memory is excluded*/
   readOnlyMemoryException             = 6,     /*memory is read-only*/
   unresolvablePageFaultException      = 7,     /*unresolvable page fault*/
   privilegeViolationException          = 8,      /*privilege violation*/
   traceException                       = 9,      /*trace*/
   instructionBreakpointException       = 10,     /*instruction breakpoint*/
   dataBreakpointException              = 11,     /*data breakpoint*/
   integerException                     = 12,     /*unused*/
   floatingPointException               = 13,     /*floating point*/
   stackOverflowException               = 14,     /*stack overflow*/
   terminationException                 = 15      /*task is being terminated*/
};
```

Constant descriptions

`unknownException`
: Unknown kind of exception. This exception code is defined for completeness only; it is never actually passed to an exception handler.

`illegalInstructionException`
: Illegal instruction exception. The processor attempted to decode an instruction that is either illegal or unimplemented.

`trapException`
: Unknown trap type exception. The processor decoded a trap type instruction that is not used by the system software.

`accessException`
: Memory access exception. A memory reference resulted in a page fault because the physical address is not accessible.

`unmappedMemoryException`
: Unmapped memory exception. A memory reference was made to an address that is unmapped.

`excludedMemoryException`
: Excluded memory exception. A memory reference was made to an excluded address.

`readOnlyMemoryException`
: Read-only memory exception. A memory reference was made to an address that cannot be written to.

`unresolvablePageFaultException`
: Unresolvable page fault exception. A memory reference resulted in a page fault that could not be resolved. The `theError` field of the memory exception record contains a status value indicating the reason for this unresolved page fault.

`privilegeViolationException`
: Privilege violation exception. The processor decoded a privileged instruction but was not executing in the privileged mode.

`traceException`
: Trace exception. This exception is used by debuggers to support single-step operations.

`instructionBreakpointException`
: Instruction breakpoint exception. This exception is used by debuggers to support breakpoint operations.

`dataBreakpointException`
: Data breakpoint exception. This exception is used by debuggers to support breakpoint operations.

`integerException`
: Integer exception. This exception is not used by PowerPC processors.

`floatingPointException`
: Floating-point arithmetic exception. The floating-point processor has exceptions enabled and an exception has occurred. (This exception is not used in the first version of the system software for PowerPC processor-based Macintosh computers.)

`stackOverflowException`
: Stack overflow exception. The stack limits have been exceeded and the stack cannot be expanded. (This exception is not used in the first version of the system software for PowerPC processor-based Macintosh computers.)

`terminationException`
: Termination exception. The task is being terminated. (This exception is not used in the first version of the system software for PowerPC processor-based Macintosh computers.)

Memory Reference Kinds

For each memory-related exception, the Exception Manager returns a memory exception record. The `theReference` field of that record contains a memory reference code that indicates the kind of memory operation that caused the exception.

```
enum {
   /*memory reference codes*/
   writeReference              = 0,     /*write operation*/
   readReference               = 1,     /*read operation*/
   fetchReference              = 2      /*fetch operation*/
};
```

Constant descriptions

`writeReference`	The operation was an attempt to write data to memory.
`readReference`	The operation was an attempt to read data from memory.
`fetchReference`	The operation was an attempt to fetch a processor instruction. (Not all processors are able to distinguish read operations from fetch operations. As a result, fetch operation failures might instead be reported as failed read operations.)

Data Structures

This section describes the data structures provided by the Exception Manager.

Machine Information Records

The Exception Manager uses a machine information record to encode the state of the special-purpose registers at the time an exception occurs. A machine information record is defined by the `MachineInformation` data type.

```
struct MachineInformation {
   UnsignedWide         CTR;  /*Count Register*/
   UnsignedWide         LR;   /*Link Register*/
   UnsignedWide         PC;   /*Program Counter Register*/
   unsigned long        CR;   /*Condition Register*/
   unsigned long        XER;  /*Fixed-Point Exception Register*/
   unsigned long        MSR;  /*Machine State Register*/
};
typedef struct MachineInformation MachineInformation;
```

Note
The fields `CTR`, `LR`, and `PC` are declared as the 64-bit type `UnsignedWide` to allow compatibility with 64-bit processors. On 32-bit processors, the register values are returned in the low-order 32 bits. The high-order 32 bits are undefined. ◆

Field descriptions

`CTR`	The contents of the Count Register (CTR).
`LR`	The contents of the Link Register (LR).
`PC`	The contents of the Program Counter Register (PC).
`CR`	The contents of the Condition Register (CR).
`XER`	The contents of the Fixed-Point Exception Register (XER).
`MSR`	The contents of the Machine State Register (MSR).

IMPORTANT
The fields of a machine information record are aligned in memory in accordance with 680x0 alignment conventions. ▲

Register Information Records

The Exception Manager uses a register information record to encode the state of the general-purpose registers at the time an exception occurs. A register information record is defined by the `RegisterInformation` data type.

```
struct RegisterInformation {
   UnsignedWide         R0;
   UnsignedWide         R1;
   UnsignedWide         R2;
   UnsignedWide         R3;
   UnsignedWide         R4;
   UnsignedWide         R5;
   UnsignedWide         R6;
   UnsignedWide         R7;
   UnsignedWide         R8;
   UnsignedWide         R9;
   UnsignedWide         R10;
   UnsignedWide         R11;
   UnsignedWide         R12;
   UnsignedWide         R13;
   UnsignedWide         R14;
   UnsignedWide         R15;
   UnsignedWide         R16;
   UnsignedWide         R17;
   UnsignedWide         R18;
   UnsignedWide         R19;
   UnsignedWide         R20;
   UnsignedWide         R21;
   UnsignedWide         R22;
   UnsignedWide         R23;
   UnsignedWide         R24;
   UnsignedWide         R25;
   UnsignedWide         R26;
   UnsignedWide         R27;
   UnsignedWide         R28;
   UnsignedWide         R29;
   UnsignedWide         R30;
   UnsignedWide         R31;
};
typedef struct RegisterInformation RegisterInformation;
```

Field descriptions

`R0`	The contents of general-purpose register GPR0.
`R1`	The contents of general-purpose register GPR1.
`R2`	The contents of general-purpose register GPR2.
`R3`	The contents of general-purpose register GPR3.
`R4`	The contents of general-purpose register GPR4.
`R5`	The contents of general-purpose register GPR5.

`R6`	The contents of general-purpose register GPR6.
`R7`	The contents of general-purpose register GPR7.
`R8`	The contents of general-purpose register GPR8.
`R9`	The contents of general-purpose register GPR9.
`R10`	The contents of general-purpose register GPR10.
`R11`	The contents of general-purpose register GPR11.
`R12`	The contents of general-purpose register GPR12.
`R13`	The contents of general-purpose register GPR13.
`R14`	The contents of general-purpose register GPR14.
`R15`	The contents of general-purpose register GPR15.
`R16`	The contents of general-purpose register GPR16.
`R17`	The contents of general-purpose register GPR17.
`R18`	The contents of general-purpose register GPR18.
`R19`	The contents of general-purpose register GPR19.
`R20`	The contents of general-purpose register GPR20.
`R21`	The contents of general-purpose register GPR21.
`R22`	The contents of general-purpose register GPR22.
`R23`	The contents of general-purpose register GPR23.
`R24`	The contents of general-purpose register GPR24.
`R25`	The contents of general-purpose register GPR25.
`R26`	The contents of general-purpose register GPR26.
`R27`	The contents of general-purpose register GPR27.
`R28`	The contents of general-purpose register GPR28.
`R29`	The contents of general-purpose register GPR29.
`R30`	The contents of general-purpose register GPR30.
`R31`	The contents of general-purpose register GPR31.

IMPORTANT

The fields of a register information record are aligned in memory in accordance with 680x0 alignment conventions. ▲

Floating-Point Information Records

The Exception Manager uses a floating-point information record to encode the state of the floating-point unit at the time an exception occurs. A floating-point information record is defined by the `FPUInformation` data type.

```
struct FPUInformation {
   UnsignedWide         Registers[32]; /*FPU registers*/
   unsigned long        FPSCR;         /*status/control reg*/
};
typedef struct FPUInformation FPUInformation;
```

Field descriptions

`Registers`	The contents of the 32 floating-point registers. This array is zero-based; for example, the contents of FPR0 are accessed as `Registers[0]`.
`FPSCR`	The contents of the Floating-Point Status and Control Register (FPSCR).

IMPORTANT

The fields of a floating-point information record are aligned in memory in accordance with 680x0 alignment conventions. ▲

Memory Exception Records

The Exception Manager uses a memory exception record to present additional information about an exception that occurs as the result of a failed memory reference. A memory exception record is defined by the `MemoryExceptionInformation` data type.

```
struct MemoryExceptionInformation {
   AreaID                          theArea;
   LogicalAddress                  theAddress;
   OSStatus                        theError;
   MemoryReferenceKind             theReference;
};
typedef struct MemoryExceptionInformation MemoryExceptionInformation;
```

Field descriptions

`theArea` The area containing the logical address of the exception. When the memory reference that caused the exception is to an unmapped range of the logical address space, this field contains the value `kNoAreaID`.

`theAddress` The logical address of the exception.

`theError` A status value. When the exception kind is `unresolvablePageFaultException`, this field contains a value that indicates the reason the page fault could not be resolved.

`theReference` The type of memory reference that caused the exception. This field contains one of these constants:

```
enum {
   writeReference = 0,  /*write operation*/
   readReference  = 1,  /*read operation*/
   fetchReference = 2   /*fetch operation*/
};
```

See “Memory Reference Kinds” on page 4-11 for a description of these constants.

IMPORTANT

The fields of a memory exception record are aligned in memory in accordance with 680x0 alignment conventions. ▲

Exception Information Records

The Exception Manager passes an exception information record to your exception handler whenever your handler is called as the result of some exception. The exception information record indicates the nature of the exception and provides other information that might be useful to your handler. An exception information record is defined by the `ExceptionInformation` data type.

```
struct ExceptionInformation {
   ExceptionKind                    theKind;
   MachineInformation               *machineState;
   RegisterInformation              *registerImage;
   FPUInformation                   *FPUImage;
   union {
      MemoryExceptionInformation    *memoryInfo;
   } info;
};
typedef struct ExceptionInformation ExceptionInformation;
```

Field descriptions

`theKind`	An exception code indicating the kind of exception that occurred. See “Exception Kinds” on page 4-9 for a list of the available exception codes.
`machineState`	The state of the machine at the time the exception occurred. See “Machine Information Records” on page 4-12 for details on the `MachineInformation` data type.
`registerImage`	The contents of the general-purpose registers at the time the exception occurred. See “Register Information Records” on page 4-12 for details on the `RegisterInformation` data type.
`FPUImage`	The state of the floating-point processor at the time the exception occurred. See “Floating-Point Information Records” on page 4-14 for details on the `FPUInformation` data type.
`memoryInfo`	The logical address of the location in memory that triggered the exception.

IMPORTANT

The fields of an exception information record are aligned in memory in accordance with 680x0 alignment conventions. ▲

Exception Manager Routines

You can use the Exception Manager's `InstallExceptionHandler` routine to install an exception handler or to remove an existing exception handler.

InstallExceptionHandler

You can use the `InstallExceptionHandler` function to install an exception handler.

```
extern ExceptionHandler InstallExceptionHandler
                                   (ExceptionHandler theHandler);
```

`theHandler`
The address of the exception handler to be installed.

DESCRIPTION

The `InstallExceptionHandler` function installs the exception handler specified by the `theHandler` parameter. That handler replaces any existing exception handler associated with the current execution context. The newly installed handler remains active until you install some other handler or until you remove the current handler by calling `InstallExceptionHandler` with `theHandler` set to `nil`.

IMPORTANT
The `theHandler` parameter must be the address of a transition vector for the exception handler, not a universal procedure pointer. ▲

The `InstallExceptionHandler` function returns the address of any existing exception handler as its function result. If there is no exception handler in place for the current execution context, `InstallExceptionHandler` returns `nil`.

SPECIAL CONSIDERATIONS

The `InstallExceptionHandler` function is available to any code executing in the PowerPC native environment. You do not need to call it if your application or other software exists as 680x0 code and hence executes under the 68LC040 Emulator on PowerPC processor-based Macintosh computers.

Application-Defined Routines

This section describes exception handlers, routines that you install using the `InstallExceptionHandler` routine to handle specific types of exceptions.

MyExceptionHandler

An exception handler should have this prototype:

```
OSStatus MyExceptionHandler (ExceptionInformation *theException);
```

`theException`
: The address of an exception information block describing the exception that triggered the exception handler.

DESCRIPTION

You pass the address of your `MyExceptionHandler` routine to the Exception Manager's `InstallExceptionHandler` function. The Exception Manager subsequently calls your exception handler for all exceptions that arise in your application's context that are not intercepted by the Operating System.

Your exception handler can take whatever steps are necessary to handle the exception or to correct the error or special condition that caused the exception. If your handler is successful, it should return the `noErr` result code. If you pass back `noErr`, the Exception Manager restores the machine state to the state contained in the exception information record pointed to by the `theException` parameter and resumes execution.

If your handler is not able to handle the exception, it should return some other result code. However, if your handler returns a nonzero result code, the current application is likely to be terminated by the Process Manager.

An exception handler uses the same stack that is active at the time an exception occurs. To ensure that no stack data is destroyed, the Exception Manager advances the stack pointer prior to calling the exception handler.

SPECIAL CONSIDERATIONS

An exception handler must follow the same general guidelines as other kinds of asynchronous software. For instance, it cannot cause memory to be purged or compacted, and it should not use any handles that are not locked. See *Inside Macintosh: Processes* for a description of the restrictions applying to interrupt tasks and other asynchronous software.

An exception handler must be reentrant if it can itself generate exceptions.

SEE ALSO

See "Writing an Exception Handler" on page 4-7 for more information about writing an exception handler.

Summary of the Exception Manager

C Summary

Constants

```
enum {
   /*exception codes*/
   unknownExceptionrecords                   = 0,     /*unknown exception type*/
   illegalInstructionException          = 1,     /*illegal instruction*/
   trapException                        = 2,     /*unknown trap type*/
   accessException                      = 3,     /*failed memory access*/
   unmappedMemoryException              = 4,     /*memory is unmapped*/
   excludedMemoryException              = 5,     /*memory is excluded*/
   readOnlyMemoryException              = 6,     /*memory is read-only*/
   unresolvablePageFaultException       = 7,     /*unresolvable page fault*/
   privilegeViolationException          = 8,     /*privilege violation*/
   traceException                       = 9,     /*trace*/
   instructionBreakpointException       = 10,    /*instruction breakpoint*/
   dataBreakpointException              = 11,    /*data breakpoint*/
   integerException                     = 12,    /*unused*/
   floatingPointException               = 13,    /*floating point*/
   stackOverflowException               = 14,    /*stack overflow*/
   terminationException                 = 15     /*task is being terminated*/
};

enum {
   /*memory reference codes*/
   writeReference                       = 0,     /*write operation*/
   readReference                        = 1,     /*read operation*/
   fetchReference                       = 2      /*fetch operation*/
};
```

Data Types

```
typedef unsigned long         ExceptionKind;    /*kind of exception*/

typedef unsigned long         MemoryReferenceKind;
typedef void                  *Ref;

typedef Ref                   AreaID;

typedef Ref                   LogicalAddress;

struct UnsignedWide {
   unsigned long              hi;
   unsigned long              lo;
};
typedef struct UnsignedWide UnsignedWide;

struct RegisterInformation {
   UnsignedWide         R0;
   UnsignedWide         R1;
   UnsignedWide         R2;
   UnsignedWide         R3;
   UnsignedWide         R4;
   UnsignedWide         R5;
   UnsignedWide         R6;
   UnsignedWide         R7;
   UnsignedWide         R8;
   UnsignedWide         R9;
   UnsignedWide         R10;
   UnsignedWide         R11;
   UnsignedWide         R12;
   UnsignedWide         R13;
   UnsignedWide         R14;
   UnsignedWide         R15;
   UnsignedWide         R16;
   UnsignedWide         R17;
   UnsignedWide         R18;
   UnsignedWide         R19;
   UnsignedWide         R20;
   UnsignedWide         R21;
   UnsignedWide         R22;
   UnsignedWide         R23;
   UnsignedWide         R24;
   UnsignedWide         R25;
   UnsignedWide         R26;
   UnsignedWide         R27;
   UnsignedWide         R28;
   UnsignedWide         R29;
   UnsignedWide         R30;
   UnsignedWide         R31;
};
typedef struct RegisterInformation RegisterInformation;

typedef long                  OSStatus;

typedef OSStatus (*ExceptionHandler) (ExceptionInformation *theException);

struct MachineInformation {
   UnsignedWide                    CTR;  /*Count Register*/
   UnsignedWide                    LR;   /*Link Register*/
   UnsignedWide                    PC;   /*Program Counter Register*/
   unsigned long                   CR;   /*Condition Register*/
   unsigned long                   XER;  /*Fixed-Point Exception Register*/
   unsigned long                   MSR;  /*Machine State Register*/
};
typedef struct MachineInformation MachineInformation;

struct FPUInformation {
   UnsignedWide                    Registers[32]; /*FPU registers*/
   unsigned long                   FPSCR;         /*status/control reg*/
};
typedef struct FPUInformation FPUInformation;

struct MemoryExceptionInformation {
   AreaID                          theArea;
   LogicalAddress                  theAddress;
   OSStatus                        theError;
   MemoryReferenceKind             theReference;
};
typedef struct MemoryExceptionInformation MemoryExceptionInformation;

struct ExceptionInformation {
   ExceptionKind                   theKind;
   MachineInformation              *machineState;
   RegisterInformation             *registerImage;
   FPUInformation                  *FPUImage;
   union {
      MemoryExceptionInformation   *memoryInfo;
   } info;
};
typedef struct ExceptionInformation ExceptionInformation;
```

Exception Manager Routines

Installing Exception Handlers

```
extern ExceptionHandler InstallExceptionHandler
                            (ExceptionHandler theHandler);
```

Application-Defined Routines

Exception Handlers

```
OSStatus MyExceptionHandler
                            (ExceptionInformation *theException);
```