Portable Systems Group

Windows NT Exception Handling Specification

Author: *David N. Cutler*

Original Draft 1.0, May 22, 1989
Revision 1.1, June 2, 1989
Revision 1.2, June 6, 1989
Revision 1.3, August, 4, 1989
Revision 1.4, August, 15, 1989
Revision 1.5, November 7, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the exception handling capabilities of **Windows NT**. An *exception* is an event that occurs during the execution of a program which requires the execution of software outside the normal flow of control.

Exceptions can result from the execution of certain instruction sequences, in which case they are initiated by hardware. Other conditions may arise as the result of the execution of a software routine (e.g., an invalid parameter value), and are therefore initiated explicitly by software.

When an exception is initiated, a systematic search is performed in an attempt to find an *exception handler* that will dispose of (handle) the exception.

An exception handler is a function written to explicitly deal with the possibility that an exception may occur in a certain sequence of code.

Exception handlers are declared in a language-specific syntax and are associated with a specific scope of code. The scope may be a block, a set of nested blocks, or an entire procedure or function.

Microsoft compilers for **Windows NT** adhere to a common calling standard which enables exception handlers to be *established* and *disestablished* in a very efficient manner.

> \ *The initial hardware target for* **Windows NT** *is the* **Intel i860***, and therefore, the* **Microsoft** *C compiler for the* **i860** *will be the first compiler that conforms to the required calling standard. As* **Windows NT** *is ported to other architectures, compilers will be required to implement a calling standard that is functional enough to support the* **Windows NT** *exception handling capabilities.* \

Exception handling capabilities are an integral and pervasive part of the **Windows NT** system. They enable a very robust implementation of the system software. It is envisioned that ISVs, application writers, and third-party compiler writers will see the benefits of exception handling capabilities and also use them in a pervasive manner.

2. Goals

The goals of the **Windows NT** exception handling capabilities are the following:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Provide a single mechanism for exception handling that is usable across all languages.
- Provide a single mechanism for the handling of hardware-, as well as software-generated exceptions.
- Provide a single exception handling mechanism that can be used by privileged, as well as nonprivileged software.
- Provide a single mechanism for the handling of exceptions and for the capabilities necessary to support sophisticated debuggers.
- Provide an exception handling architecture with the capabilities necessary to emulate the exception handling capabilities of other operating systems (e.g. OS/2 and POSIX).
- Provide an exception handling mechanism that is portable and which separates machine-dependent from machine-independent information.
- Provide an exception handling mechanism that supports the structured exception handling extensions being proposed by **Microsoft** for the C language (see Structured Exception Handling in C by Don MacLaren, May 10, 1989).

3. Exception Architecture

The overall exception architecture of **Windows NT** encompasses the process creation primitives, system service emulation subsystems, the **Microsoft** calling standard(s), and system routines that raise, dispatch, and unwind exceptions.

Two optional exception ports may be specified when a process is created. These ports are called the debugger port and the system service emulation subsystem port.

When an exception is initiated, an attempt is made to send a message to the recipient process's debugger port. If there is no debugger port, or the associated debugger does not handle the exception, then a search of the current thread's call frames is conducted in an attempt to locate an exception handler. If no frame-based handler can be found, or none of the frame-based handlers handle the exception, then another attempt is made to send a message to the recipient process's debugger port. If there is no debugger port, or the associated debugger does not handle the exception, then an attempt is made to send a message to the recipient process's system service emulation

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

subsystem port. If there is no subsystem port, or the subsystem does not handle the exception, then the system provides default handling based on the exception type.

Thus the search hierarchy is:

1. Debugger first chance
2. Frame-based handlers
3. Debugger second chance
4. Emulation subsystem

The purpose of this architecture is to provide a very robust exception architecture, while at the same time allow for the emulation of the exception handling capabilites of various operating system environments (e.g., OS/2 2.0 exception handling, POSIX signals, etc.).

Throughout this document, explanations concerning the implementation of the **Windows NT** exception architecture are given referring to the **Intel i860**. It should not be inferred that the described implementation is the only possible implementation, and in fact, the actual implementation on other hardware architectures may be different.

3.1 Frame-Based Exception Handlers

An exception handler can be associated with each call frame in the procedure call hierarchy of a program. This requires that each procedure or function that either saves nonvolatile registers or establishes an associated exception handler, have a call frame.

Microsoft compilers for **Windows NT** adhere to a standard calling convention for the construction of a call frame. A call frame for the **Intel i860** contains the following:

1. A register save mask that describes the nonvolatile registers saved in the frame. These registers are saved in a standard place relative to the frame pointer.
2. Two flags that specify whether an extended register save mask and/or exception handler address is present in the frame.
3. An optional extended register save mask that describes the volatile registers saved in the frame. These registers are saved in a standard place relative to the frame pointer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. An optional address of an exception handler that is associated with the frame.

Call frames for other architectures contain similar information. The exact details of the call frame layout are described in the **Microsoft Windows NT** calling standard(s).

3.2 Exception Dispatching

When a hardware exception occurs, the **Windows NT** trap handling software gets control and saves the hardware state of the current thread in a *context record*. The reason for the trap is determined, and an *exception record* is constructed which describes the exception and any pertinent parameters. Executive software is then called to dispatch the exception.

If the previous processor mode was kernel, then the exception dispatcher is called to search the kernel stack call frames in an attempt to locate an exception handler. If a frame-based handler cannot be located, or no frame-based handler handles the exception, then **KeBugCheck** is called to shut down system operation. Unhandled exceptions emanating from within privileged software are considered fatal bugs.

If the previous processor mode was user, then an attempt is made to send a message to the associated debugger port. This message includes the exception record and the identification of the client thread. The debugger may handle the exception (e.g., breakpoint or single step) and modify the thread state as appropriate, or not handle the exception and defer to any frame-based exception handlers found on the user stack.

If the debugger replies that it has handled the exception, then the machine state is restored and thread execution is continued. Otherwise, if the debugger replies that it has not handled the exception, or there is no debugger port, then executive software must prepare to execute the exception dispatcher in user mode.

If the debugger does not dispose of the exception, then stack space is allocated on the user stack, and both the exception record and the context record are moved to the user stack. The machine state of the thread is modified such that thread execution will resume in code that is part of the executive, but it executes in user mode.

The machine state is restored and execution of the thread is resumed in user mode within executive code that calls the exception dispatcher to search the user stack for an exception handler. If a frame-based handler handles the exception, then thread execution is continued. Otherwise, if no frame-based handler is found, or no frame-

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

based handler handles the exception, then the **NtLastChance** system service is executed.

The purpose of the **NtLastChance** system service is to provide the debugger a second chance to handle the exception and to provide the system service emulation subsystem associated with the thread's process, if any, a chance to perform any subsystem-specific exception processing. A second attempt is made to send a message to the associated debugger port. This message includes the exception record and the identification of the client thread. The debugger may handle the exception (e.g., query the user and receive a disposition) and modify the thread state as appropriate, or not handle the exception and defer to the system service emulation subsystem associated with the thread's process.

If the debugger replies that it has handled the exception, then the machine state is restored and thread execution is continued. Otherwise, if the debugger replies that it has not handled the exception, or there is no debugger port, then an attempt is made to send a message to the associated subsystem. This message includes the exception record and the identification of the client thread. The subsystem may handle the exception and modify the thread state as appropriate, or not handle the exception and defer to any default handling supplied by the executive.

If the subsystem replies that it has handled the exception, then the machine state is restored and thread execution is continued. Otherwise, the executive provides default handling of the exception, which is most cases causes the thread to be terminated.

3.3 Exception Handling and Unwind

During the dispatching of an exception, each frame-based handler is called specifying the associated exception and context records as parameters. The exception handler can handle the exception and continue execution, not handle the exception and continue the search for an exception handler, or handle the exception and initiate an unwind operation.

Handling an exception may be as simple as noting an error and setting a flag that will be examined later, printing a warning or error message, or taking some other overt action. If execution can be continued, then it may be necessary to change the machine state by modifying the context record (e.g., advance the continuation instruction address).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If execution can be continued, then the exception handler returns to the exception dispatcher with a status code that specifies that execution should be continued. Continuing execution causes the exception dispatcher to stop its search for an exception handler. The machine state from the context record is restored and execution is continued accordingly.

If execution of the thread cannot be continued, then the exception handler usually initiates an unwind operation by calling a system-supplied function specifying a target call frame and a continuation address. The unwind function walks the stack backwards searching for the target call frame. As it walks the stack, the unwind function calls each exception handler that is encountered to allow it to perform any cleanup actions that may be necessary (e.g., release a semaphore, etc.). When the target call frame is reached, the machine state is restored and execution is continued at the specified address.

3.4 Exception Record

An *exception record* describes an exception and its associated parameters. The same structure is used for both hardware-, and software-generated exceptions.

An exception record has the following structure:

Exception Record Structure

NTSTATUS *ExceptionCode* - The status code that specifies the reason for the exception.

ULONG *ExceptionFlags* - A set of flags that describes attributes of the exception.

Exception Flags

EXCEPTION_NONCONTINUABLE - The exception is not continuable, and any attempt to continue will cause the exception **STATUS_NONCONTINUABLE_EXCEPTION** to be raised.

EXCEPTION_UNWINDING - The exception record describes an exception for which an unwind is in progress.

EXCEPTION_EXIT_UNWIND - The exception record describes an exception for which an exit unwind is in progress.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

EXCEPTION_STACK_INVALID - The user stack was not within the limits specified by the Thread Environment Block (**TEB**) when the exception was raised in user mode. Alternately, during the trace backwards through the call frames on the user (or kernel) stack, a call frame was encountered that was not within the stack limits specified by the **TEB** (or within the kernel stack limits), or a call frame was encountered that was unaligned.

EXCEPTION_NESTED_CALL - The exception record describes an exception that was raised while the current exception handler was active, i.e., a nested exception is in progress and the current handler was also called to handle the previous exception.

PEXCEPTION_RECORD *ExceptionRecord* - An optional pointer to an associated exception record. Exception records can be chained together to provide additional information when nested exceptions are raised.

PVOID *ExceptionAddress* - The instruction address at which the hardware exception occurred or the address from which the software exception was raised.

ULONG *NumberParameters* - The number of additional parameters that further describe the exception and immediately follow this parameter in the exception record.

ULONG *ExceptionInformation[NumberParameters]* - Additional information that describes the exception.

The *EXCEPTION_NONCONTINUABLE* bit in the exception flags field is the only flag that can be set by the user. The remaining flags are set by system supplied software as the result of dispatching an exception or the unwinding of call frames.

3.5 Exception Context

A *context record* describes the machine state at the time an exception occurred. This record is hardware architecture dependent and is not portable. Therefore, in general, software should not use the information contained is this record. Hardware-architecture-dependent code such as math libraries, however, can make use of this information to optimize certain operations.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

For a hardware-initiated exception, the context record contains the complete machine state at the time of the exception. For a software-initiated exception, the context record contains the machine state at the time the exception was raised by software.

The context record is constructed so that it has an identical format to the call frames generated by the **Microsoft** compilers for the **Intel i860**. The context record for the **Intel i860** has the following structure:

Context Record Structure

ULONG *ContextFlags* - A set of flags that describes which sections of the context record contain valid information.

Context Flags

CONTEXT_CONTROL - The *Psr*, *Epsr*, *Fir*, *IntR1*, *IntFp*, and *IntSp* fields of the context record are valid.

CONTROL_FLOATING_POINT - The *FltF2...FltF31* and *Fsr* fields of the context record are valid.

CONTEXT_INTEGER - The *IntR4...IntR31* fields of the context record are valid.

CONTEXT_PIPELINE - The *AddStageX*, *MulStageX*, *FldStageX*, *IntResult*, *Kr*, *Ki*, *Merge*, *T*, *Fsr1, Fsr2*, *and Fsr3* fields of the context record are valid.

ULONG *Fsr* - The contents of the floating point status register (FSR) at the time of the exception.

UQUAD *AddStage1*, *AddStage2*, *AddStage3* - Stages 1 - 3 of the floating point addition pipeline.

UQUAD *MulStage1*, *MulStage2*, *MulStage3* - Stages 1 - 3 of the floating point multiplication pipeline.

UQUAD *FldStage1*, *FldStage2*, *FldStage3* - Stages 1 - 3 of the floating point load pipeline.

UQUAD *IntResult* - The integer result of the graphics pipeline.

UQUAD *Kr* - The contents of the **KR** register.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

UQUAD *Ki* - The contents of the **KI** register.

UQUAD *Merge* - The contents of the **MERGE** register.

UQUAD *T* - The contents of the **T** register.

ULONG *Fir* - The continuation instruction pointer.

ULONG *Fsr1*, *Fsr2*, *Fsr3* - The contents of the floating status register (FSR) for stages 1 - 3 of the pipeline.

ULONG *IntR4...IntR31* - The contents of the integer registers **r4** - **r31**.

UQUAD *FltF2...FltF31* - The contents of the floating point registers **f2** - **f31**.

ULONG *IntSp* - The contents of the stack pointer at the time of the exception.

ULONG *ExtendedSaveMask* - The extended register save mask that specifies that register *Int16...IntR31* are saved in the record.

ULONG *Handler* - The address of the associated exception handler.

ULONG *RegisterSaveMask* - The standard register save mask that specifies that registers *IntR4...IntR15* and *FltF2...FltF31* are saved in the record.

ULONG *IntFp* - The contents of the frame pointer at the time of the exception.

ULONG *IntR1* - The contents of the register **R1** (return address) at the time of the exception.

ULONG - *Psr* - The processor status (PSR) at the time of the exception.

ULONG - *Epsr* - The extended processor status (EPSR) at the time of the exception.

4. Hardware-Defined Exceptions

Hardware-defined exceptions are initiated by the executive when a particular kind of fault condition is encountered as the result of instruction execution, e.g., an integer overflow. System software collects the information necessary to initiate the exception and then calls a routine that routes the exception to the appropriate exception handler.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following sections describe the various hardware-defined exceptions in a machine-independent format. For each exception, the exception status code and any additional parameters are specified. These values are placed in the exception record when the particular type of exception is generated. In addition, any pertinent **Intel i860**-dependent information is also provided.

Not all hardware architectures generate all the various exceptions that are defined. Each port of **Windows NT** to a new hardware architecture requires a mapping of the hardware-defined exceptions onto the machine-independent format given below.

> \ *The following sections must be carefully examined to ensure that they represent a machine-independent description for x86, as well as* **i860***, exceptions.* \

4.1 Access Violation

An access violation exception is generated when an attempt is made to load or store data from/to a location that is not accessible to the current process, or when an attempt is made to execute an instruction that is not accessibile to the current process.

Exception Code: **STATUS_ACCESS_VIOLATION**

Additional Parameters: 2

> Read/Write - A value of zero signifies a read; a value of one signifies a write.
>
> Virtual Address - The virtual address of the data that is not accessible.

4.2 Breakpoint

A breakpoint exception occurs when a breakpoint instruction is executed, or a hardware-defined breakpoint is encountered (e.g. an address in a breakpoint register). This exception is intended for use by debuggers.

Exception Code: **STATUS_BREAKPOINT**

Additional Parameters: 1

> Read/Write - A value of zero signifies a read; a value of one signifies a write.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

i860 Implementation: The execution of a **TRAP r30,r29,r0** instruction, or a match with the address in the breakpoint register causes a breakpoint exception on the **Intel i860**.

4.3 Data-Type Misalignment

A data-type misalignment exception is generated when an attempt is made to load or store data from/to an address that is not naturally aligned, on a hardware architecture that does not provide alignment hardware. For example, 16-bit entities must be aligned on two-byte boundaries, 32-bit entities must be aligned on four-byte boundaries, etc.

Exception Code: **STATUS_DATATYPE_MISALIGNMENT**

Additional Parameters: 3

Read/Write - A value of zero signifies a read; a value of one signifies a write.

Data-type Mask - A data-type mask that specifies how many low-address bits must be zero. For example, the data-type mask for a 16-bit entity is one, a 32-bit entity three, etc.

Virtual Address - The virtual address of the misaligned data.

4.4 Floating Divide By Zero

A floating divide by zero exception is generated when an attempt is made to divide a floating point dividend by a floating point divisor of zero.

Exception Code: **STATUS_FLOATING_DIVIDE_BY_ZERO**

Additional Parameters: None

4.5 Floating Overflow

A floating overflow exception is generated when the resulting exponent of a floating point operation is greater than the magnitude allowed for the respective floating point data type.

Exception code: **STATUS_FLOATING_OVERFLOW**

Additional Parameters: None

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.6 Floating Underflow

A floating underflow exception is generated when the resulting exponent of a floating point operation is less than the magnitude provided for the respective floating point data type.

Exception Code: **STATUS_FLOATING_UNDERFLOW**

Additional Parameters: None

4.7 Floating Reserved Operand

A floating reserved operand exception is generated when one or more of the source operands in a floating point operation have a format that is reserved.

Exception Code: **STATUS_FLOATING_RESERVED_OPERAND**

Additional Parameters: None

4.8 Illegal Instruction

An illegal instruction exception is generated when an attempt is made to execute an instruction whose operation is not defined for the host machine architecture.

Exception Code: **STATUS_ILLEGAL_INSTRUCTION**

Additional Parameters: None

i860 Implementation: The execution of a **TRAP** instruction other than **TRAP r30,r29,r0** or **TRAP r30,r28,r0** or **TRAP r30,r27,r0** causes an illegal instruction exception.

4.9 Privileged Instruction

A privileged instruction exception is generated when an attempt is made to execute an instruction whose operation is not allowed in current machine mode (e.g., an attempt to execute an instruction from user mode that is only allowed in kernel mode).

Exception Code: **STATUS_PRIVILEGED_INSTRUCTION**

Additional Parameters: None

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.10 Invalid Lock Sequence

An invalid lock sequence exception is generated when an attempt is made to execute an operation, within an interlocked section of code, such that the sequence is invalid for the host machine architecture.

Exception Code: **STATUS_INVALID_LOCK_SEQUENCE**

Additional Parameters: None

i860 Implementation: Exceeding the 32-instruction limit within a lock sequence, an attempt to execute a **TRAP** instruction within a lock sequence, or an attempt to execute an **INTOVR** instruction within a lock sequence causes an invalid lock sequence exception.

4.11 Integer Divide By Zero

An integer divide-by-zero exception is generated when an attempt is made to divide an integer dividend by an integer divisor of zero.

Exception Code: **STATUS_INTEGER_DIVIDE_BY_ZERO**

Additional Parameters: None

4.12 Integer Overflow

An integer overflow exception is generated when the result of an integer operation causes a carry out of the the most significant bit of the result, which is not the same as the carry into of the most significant bit of the result. For example, the addition of two positive integers that produces a negative result.

Exception Code: **STATUS_INTEGER_OVERFLOW**

Additional Parameters: None

i860 Implementation: The execution of an **INTOVR** instruction when **OF** set in **EPSR** causes an integer overflow exception. The **OF** bit in EPSR is cleared prior to initiating this exception.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.13 Single Step

A single step exception is generated when a trace trap or other single instruction execution mechanism signals that one instruction has been executed. This exception is intended for use by debuggers.

Exception Code: **STATUS_SINGLE_STEP**

Additional Parameters: None

i860 Implementation: The execution of a **TRAP r30,r28,r0** instruction causes a single step exception.

5. Windows NT Software-Defined Exceptions

Windows NT software-defined exceptions are explicitly raised by system software when certain conditions are encountered, e.g., a page file read error. System software collects the information necessary to initiate the exception and then calls a routine that routes the exception to the appropriate exception handler.

5.1 Guard Page Violation

A guard page violation exception is generated when an attempt is made to load or store data from/to a location that is contained within a guard page. Memory management software immediately turns the guard page into a demand zero page and initiates a guard page violation exception.

Exception Code: **STATUS_GUARD_PAGE_VIOLATION**

Additional Parameters: 2

Read/Write - A value of zero signifies a read; a value of one signifies a write.

Virtual Address - The virtual address of the data within a guard page.

5.2 Page Read Error

A page read error exception is generated when an attempt is made to read a page into memory and an I/O error is encountered.

Exception Code: **STATUS_IN_PAGE_ERROR**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Additional Parameters: 1

Virtual Address - A virtual address within the page that was being read.

5.3 Paging File Quota Exceeded

A page file quota exceeded exception is generated when an attempt is made to commit backing store space for a page that is being removed from a process's working set.

Exception Code: **STATUS_PAGEFILE_QUOTA**

Additional Parameters: 1

Virtual Address - A virtual address within the page that was being read.

6. Standard Exception Handling

Standard exception handling is provided for some exceptions in which it is most likely that the user will select the default handling as the first resort, rather than wait until all other handlers have been given an opportunity to handle the exception. This enables the fault to be handled in the most efficient manner.

This capability is provided in **Windows NT** for alignment faults and IEEE floating point faults.

6.1 Alignment Faults

Standard handling of alignment faults ... TBS

6.2 IEEE Floating Faults

Standard handling of IEEE faults ... TBS

7. Exception Handling Interfaces

Several interfaces are supplied by the **Windows NT** system to implement the exception handling architecture described above. Some of these interfaces are intended for use only by the exception handling components themselves, while others are available to user-level software. The following subsections describe the exception handling APIs that are provided by **Windows NT**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.1 Exception Dispatcher

The exception dispatcher is responsible for searching the stack for frame-based exception handlers. There is a single exception dispatcher and it is responsible for dispatching both hardware-, and software-generated exceptions.

The exception dispatcher can be invoked with the **RtlpDispatchException** function:

BOOLEAN
RtlDispatchException (
    IN PEXCEPTION_RECORD *ExceptionRecord*,
    IN PCONTEXT *ContextRecord*
    );

Parameters:

ExceptionRecord - A pointer to an exception record that describes the exception, and the parameters of the exception, that has been raised.

ContextRecord - A pointer to a context record that describes the machine state at the time the exception occurred.

The exception dispatcher walks backward through the call frame hierarchy attempting to find an exception handler that will handle the exception. As each handler is encountered, it is called specifying the exception record, the context record, the address of the call frame of the establisher of the handler, and whether the handler is being called recursively or not, as parameters.

The exception handler may handle the exception or request that the scan of call frames be continued. As each step backwards is made in the call hierarchy, a check is made to ensure that the previous call frame address is within the current thread's stack limits and is aligned. If the stack is not within limits or is unaligned, then the *EXCEPTION_STACK_INVALID* flag is set in the exception flags field and the **NtLastChance** system service is called to finish processing of the exception. Otherwise, the previous frame is examined to determine if it specifies an exception handler.

The exception dispatcher is called by **RtlRaiseException** and by the executive code that processes hardware-generated exceptions.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.2 Exception Handler

An exception handler is usually called by the exception dispatcher, specifying parameters that describe the exception and the environment in which the exception handler was established. Exception handlers, however, are also called during an unwind operation and are given a chance to clean up data structures, deallocate resources, or do any other operations that are necessary to unwind the establisher's call frame.

The *EXCEPTION_UNWINDING*, *EXCEPTION_EXIT_UNWIND*, and *EXCEPTION_NESTED_CALL* flags in the exception record determine how an exception handler is being called. These flags are set by the exception dispatcher and the unwind function. If both the *EXCEPTION_UNWINDING* and *EXCEPTION_EXIT_UNWIND* flags are clear, then the exception handler is being called to handle an exception. Otherwise, an unwind operation is in progress, and the exception handler is being called to perform any necessary cleanup operations. If the exception handler is being called to handle an exception, then the *EXCEPTION_NESTED_CALL* flag determines whether a nested exception is in progress (i.e., another exception was raised in the containing scope before the previous exception was disposed of).

An exception handler has the following type definition:

typedef
EXCEPTION_DISPOSITION
(***PEXCEPTION_ROUTINE**) (
    IN PEXCEPTION_RECORD *ExceptionRecord*,
    IN PVOID *EstablisherFrame*,
    IN OUT PCONTEXT *ContextRecord*,
    IN OUT PVOID *DispatcherContext*
    );

Parameters:

ExceptionRecord - A pointer to an exception record that describes the exception and the parameters of the exception.

EstablisherFrame - A pointer to the call frame of the establisher of the exception handler.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ContextRecord - A pointer to a context record that describes the machine state at the time the exception occurred.

DispatcherContext - A pointer to a record that receives state information on nested exceptions and collided unwinds.

When an exception handler is called to handle an exception, it has several options for how it processes an exception:

1. It can handle the exception, provide any fixup that is necessary by modifying the context record, and then continue execution of the program at the point of the exception by returning a disposition value of *ExceptionContinueExecution*.

2. It can handle the exception, determine that execution cannot be continued, and initiate an unwind operation.

3. It can decide that it cannot handle the exception and return a disposition value of *ExceptionContinueSearch*, which causes the exception dispatcher to continue the search for an exception handler.

When an exception handler is called during an unwind operation, it also has several options for how it processes the call:

1. It can perform any necessary cleanup operations and return a disposition value of *ExceptionContinueSearch*.

2. It can perform any necessary cleanup operations and initiate another unwind operation to a different target.

3. It can restore the machine state from the context record and continue execution directly.

If the exception handler belongs to the exception dispatcher itself, then it can also return a disposition value of *ExceptionNestedException* when it is called to handle an exception. Likewise, if the exception handler belongs to the unwind function, then it can also return a disposition value of *ExceptionCollidedUnwind* when it is called to perform any necessary cleanup operations (i.e., an unwind is in progress). For both of these cases, the *DispatcherContext* parameter is used to return information to either the exception dispatcher or the unwind function. No other exception handler can place information in this output parameter.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If an invalid disposition value is returned by an exception handler, then the exception **STATUS_INVALID_DISPOSITION** is raised by the exception dispatcher.

The *ContextRecord* parameter is intended for use by machine-specific code that either restores the machine state during an unwind operation, or manipulates the machine state in such a way as to fix up an exception. An example of such an exception handler is the default IEEE floating point exception handler, which uses the machine state information to determine how a floating point exception should actually be handled. Another example is the fixup necessary for unaligned data references. This type of exception handler is machine specific and will generally be supplied by **Microsoft**.

When an exception handler is called to handle an exception, the context record contains the machine state at the time of the exception. However, when an exception handler is called during an unwind operation, the context record contains the machine state of the exception handler's establisher.

When a disposition value of *ExceptionContinueExecution* is returned, the exception dispatcher checks to determine if the exception is continuable. If it is not continuable (i.e., the *EXCEPTION_NONCONTINUABLE* flag is set in the exception flags field of the exception record), then the exception dispatcher raises the exception **STATUS_NONCONTINUABLE_EXCEPTION**. Otherwise, the machine state is restored and execution resumes at the point of the exception.

A disposition value of *ExceptionContinueSearch* causes the exception dispatcher or unwind function to continue its scan of call frames.

If the exception handler of the exception dispatcher is encountered during the scan for an exception handler, then it returns a disposition value of *ExceptionNestedException* and the address of the call frame that established the exception handler most recently called by the exception dispatcher. The *EXCEPTION_NESTED_CALL* flag is set in the exception flags field of the exception record for each exception handler that is called between the exception dispatcher handler and the establisher of the most recently called exception handler. It is the responsibility of the individual exception handlers themselves to determine if they can be recursively called.

The exception handler of the unwind function returns a disposition value of *ExceptionCollidedUnwind* and the target frame of the current unwind. This information is used to determine the new scope of the unwind.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.3 Raise Exception

A software exception can be raised with the **RtlRaiseException** function:

VOID
RtlRaiseException (
    IN PEXCEPTION_RECORD *ExceptionRecord*
    );

Parameters:

ExceptionRecord - A pointer to an exception record that describes the exception, and the parameters of the exception, that is raised.

Raising a software exception captures the machine state of the current thread in a context record. The *ExceptionAddress* field of the exception record is set to the caller's return address, and the exception dispatcher is then called in an attempt to locate a frame-based exception handler to handle the exception. Note that the associated debugger, if any, is not given a first chance to handle software exceptions.

If an exception handler returns a disposition value of *ExceptionContinueExecution*, then execution will return to the caller of **RtlRaiseException**. If no frame-based exception handler disposes of the exception, then **NtLastChance** is called to enable the appropriate system service emulation subsystem to perform any subsystem-specific processing.

7.4 Continuing From An Exception

Execution of a thread can be continued from the point of an exception with the **NtContinue** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
NtContinue (
    IN PCONTEXT *ContextRecord*,
    IN BOOLEAN *TestAlert*
    );

Parameters:

ContextRecord - A pointer to a context record that describes the machine state that is to be restored.

TestAlert - A boolean value that specifies whether an alert should be tested for the previous processor mode. This parameter is used for APC processing.

This function restores the machine state from the specified context record and resumes execution of the thread.

\ *Note that such a service would not normally be required. The* **Intel i860** *architecture, however, does not allow the entire machine state to be completely restored in user mode, and therefore, a system service must be called in kernel mode to perform this operation.* \

This function is called by the exception dispatcher to continue the execution of a thread when an exception handler returns a dispostion value of *ExecptionContinueExecution*.

7.5 Unwinding From An Exception

An exception can be unwound with the **RtlUnwind** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
RtlUnwind (
IN PVOID *TargetFrame* **OPTIONAL**,
IN PVOID *TargetIp* **OPTIONAL**,
IN PEXCEPTION_RECORD *ExceptionRecord* **OPTIONAL**
);

Parameters:

TargetFrame - An optional pointer to the call frame that is the target of the unwind. If this parameter is not specified, then the *EXCEPTION_EXIT_UNWIND* flag is set in the exception flags field of the exception record.

TargetIp - An optional instruction address that specifies the continuation address. This parameter is ignored if the *TargetFrame* parameter is not specified.

ExceptionRecord - An optional pointer to an exception record that is used when each exception handler is called during the unwind operation.

This function initiates an unwind of procedure call frames. The machine state at the time of the call to **RtlUnwind** is captured in a context record, the *EXCEPTION_UNWINDING* flag is set in the exception flags field of the exception record, and the *EXCEPTION_EXIT_UNWIND* flag is also set if the *TargetFrame* parameter is not specified. A backward walk through the procedure call frames is then performed to find the target of the unwind operation.

As each call frame is unwound, the machine state of the previous frame is computed by restoring any registers stored by the procedure. The previous frame is then examined to determine if it has an associated exception handler. If the call frame has an exception handler, then it is called specifying the exception record, the establisher's frame pointer, and the context record that contains the machine state of the handler's establisher. The exception handler should perform any cleanup operations that are necessary, and continue the unwind operation by returning a disposition value of *ExceptionContinueSearch*, initiating another unwind operation, or directly restoring the machine state from the context record.

Note that languages that support a termination model for exception handling (e.g., Ada, Modula-3, and the proposed extensions to **Microsoft** C) can implement this capability by unwinding to the frame of the establisher when a language-specific exception

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

handler is invoked during either an unwind operation or during the dispatching of an exception.

There is no return from a call to **RtlUnwind**. Control is either transferred to the specified instruction pointer address, or **NtLastChance** is called to perform secondary debugger processing and/or subsystem-specific default processing at the completion of the unwind operation. If **RtlUnwind** encounters an error during its processing, it raises another exception rather than return control to the caller.

If the target call frame is reached and an exit unwind is not being performed (i.e. the *TargetFrame* parameter is specified), then the computed machine state is restored from the context record and control is transfered to the address specified by the *TargetIp* parameter. Note that the stack pointer is not restored making it possible to transfer information on the stack. It is the responsibility of the code at the target address to reset the stack pointer as necessary.

If an exit unwind is being performed (i.e. the *TargetFrame* parameter is not specified), then all call frames are unwound until the base of the stack is reached. **NtLastChance** is then called to perform secondary debugger processing and/or subsystem-specific processing.

If the *ExceptionRecord* parameter is specified, then each exception handler encountered during the unwind operation is called using the specified record. If this parameter is not specified, then **RtlUnwind** constructs an exception record that specifies the exception **STATUS_UNWIND**.

During an unwind operation, it is possible for one unwind to *collide* with a previous unwind. This occurs when the scope of the second unwind overlaps the scope of the first unwind.

There are two cases to consider:

1. The target frame of the second unwind is a frame that has already been unwound by the first unwind.

2. The target frame of the second unwind occurs earlier in the call hierarchy than the target of the first unwind.

The first case is processed by unwinding call frames for the second unwind until the first call frame unwound by the first unwind is encountered. The second unwind is then

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

terminated and processing of the first unwind is continued at the point where the first unwind was interrupted by the second unwind.

The second case is processed by changing the target of the first unwind to that of the second unwind, and then applying the handling that is provided for case one.

7.6 Last Chance Exception Handling

Last chance exception handling can be invoked with the **NtLastChance** function:

NTSTATUS
NtLastChance (
    IN PEXCEPTION_RECORD *ExceptionRecord*,
    IN PCONTEXT *ContextRecord*
    );

Parameters:

ExceptionRecord - A pointer to an exception record that describes the exception, and the parameters of the exception, that has been raised.

ContextRecord - A pointer to a context record that describes the machine state at the time the exception occurred.

Last chance handling copies the exception and context records onto the kernel stack and checks to determine if a system service emulation subsystem port is associated with the thread's process. If a subsystem port is associated with the thread's process, then a message is sent to the port specifying the exception record and the identification of the client thread. Otherwise, default handling is provided for the exception.

This function is called by the exception dispatcher to perform subsystem and/or default handling for an exception that is not handled by any of the frame-based exception handlers. It is not called by any other component of the system.

Normally there is no return from the call to **NtLastChance**. However, if the context or exception record is not accessible to the calling process, then an access violation status is returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

8. OS/2 2.0 Compatibility

The OS/2 Cruiser project is currently designing a new exception handling capability for OS/2 that replaces the current **DosSetVec** interface, and which can provide the basis for frame-based exception handling.

It is desirable to be able to directly emulate the proposed exception capabilities of OS/2 with the native frame-based exception handling provided by **Windows NT**. Furthermore, it is desirable to be able to use both the OS/2 style of exception handlers in the same program as the **Windows NT** frame-based handlers without a conflict arising. Currently this is not possible without further refinement of the OS/2 proposal and the introduction of certain constraints concerning the establishment and disestablishment of OS/2 exception handlers. Other problems with the current OS/2 design include the visibility of x86 architectural features, which makes the user interface nonportable.

The following changes and restrictions need to be specified:

1. The machine-dependent state must be separated from the exception state in the OS/2 proposal so that portable software can be written that makes use of the exception handling capabilities of OS/2 on architectures other than the x86.

2. The exception information included in an exception record for OS/2 must be specified in such a way as to be portable to architectures other than the x86 (i.e., a higher level of abstraction is needed for the parameter values).

3. A restriction must be placed on the establishment and disestablishment of OS/2 exception handlers such that they are strictly frame based (i.e., an exception handler that is established in a frame must be disestablished before leaving the frame).

4. A restriction must be placed on the use of OS/2 style exception handlers in the same frame as **Windows NT** frame-based exception handlers.

5. The semantics of **DosRaiseException** must be corrected to return to the call site if a continuation status is returned by an exception handler.

If these changes and restrictions are implemented, then the **Windows NT** exception handling capabilities, with slight modification, can directly emulate the OS/2 exception handling capabilities.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

8.1 Windows NT Intel i860 Implementation

The **Intel i860 Windows NT** implementation of exception handling is frame based. Each call frame has a pointer that is dedicated to holding the address of an exception handler for the frame. Usually this is a language-supplied handler that provides whatever semantics are required to provide exception handling for the language. If there is no handler associated with the frame, then a flag is clear in the frame to signify that there is no exception handler and the dedicated pointer does not contain meaningful information. If the handler address is specified as VOID, then there is also no exception handler associated with the frame.

When an exception occurs, or an unwind operation is initiated, a backward walk through the call frames is conducted. If a call frame contains an exception handler, then it is called with the appropriate arguments.

Establishing an exception handler does not require the allocation of any heap storage, or the initialization of any data structure on the part of the user. Exception handlers are automatically disestablished upon leaving a procedure and deleting its call frame from the stack.

Unwind does not return to its caller. Rather it unwinds call frames, calling exception handlers as appropriate until the target of the unwind is reached, and then restores the machine state and transfers control to a specified destination instruction address.

8.2 OS/2 2.0 Intel x86 Implementation

The OS/2 implementation of exception handling on the **Intel x86** is list based. The head of the list is anchored in the Thread Information Block (TIB) of a thread. When an exception handler is established, a structure called an exception-handler-structure is supplied by the user, and linked into the last in, first out (LIFO) list of exception handlers. Disestablishing an exception handler removes the appropriate exception-handler-structure from the list.

The exception-handler-structure contains a link pointer and a pointer to the exception handler associated with the structure. The fields of the structure are exported to the user who is free to change the address of the exception handler while the structure is in the exception list.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

When an exception occurs, or an unwind operation is initiated, a forward walk through the exception handler list is performed. Each handler is called with the appropriate arguments.

After completing an unwind operation (no unwind is actually done), the OS/2 function returns control to the caller, which must perform a longjmp() if necessary.

OS/2 defines the following APIs for exception handling:

1. **DosSetExceptionHandler** (**exception-handler-structure*) - This function establishes an exception handler by placing the specified exception-handler-structure at the front of the exception handler list.

2. **DosUnsetExceptionHandler** (**exception-handler-structure*) - This function disestablishes an exception handler by removing the specified exception-handler-structure from the exception handler list.

3. **DosRaiseException** (**exception-structure*) - This function raises a software exception.

4. **DosUnwindException** (**exception-handler-structure*) - This function causes exception handlers to be called up, including the exception handler specified by the exception-handler-structure.

8.3 Windows NT Implementation of OS/2 Capabilities

In order to directly emulate OS/2 exception handling in **Windows NT**, the restrictions and changes described above for OS/2 must be made. Assuming these changes are made, the following paragraphs describe how **Windows NT** can directly emulate the OS/2 capabilities.

The meaning of the handler address in a call frame is expanded to be either a handler address (low-order bit is clear), or a pointer to a LIFO list of exception-handler-structures (low-order bit is set). A call frame can contain either a list head for OS/2 style exception handlers or a pointer to a single exception handler for **Windows NT** exceptions.

The function **DosSetExceptionHandler** inserts an exception-handler-structure in the LIFO list of exception handlers defined for the current call frame. If there is a Windows **NT** exception handler already established for the frame, then an attempt to insert an

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

OS/2 style handler causes the exception **STATUS_INCOMPATIBLE_EXCEPTION_HANDLER** to be raised. Otherwise, the specified exception-handler-structure is inserted at the front of the exception handler list and the low-order bit of the exception handler address is set.

The function **DosUnsetExceptionHandler** removes an exception-handler-structure from the exception list associated with the current frame. If the current frame contains a Windows **NT** exception handler, or the specified exception-handler-structure is not in the current frame's exception handler list, then the exception **STATUS_HANDLER_NOT_FOUND** is raised. Otherwise, the specified exception-handler-structure is removed from the exception handler list of the currrent frame.

The function **DosRaiseException** reformats the exception record that it is passed into the exception record expected by **RtlRaiseException**. No other processing is required. If an exception handler returns a continuation status, then control returns to the caller of **DosRaiseException**.

The function **DosUnwindException** performs a prescan of call frames in an attempt to locate the specified exception-handler-structure. The prescan is performed by walking backwards through the call frame and examining the exception handler list for each frame that contains such a list. If the specified exception-handler-structure is not found, then the exception **STATUS_HANDLER_NOT_FOUND** is raised. Otherwise, **RtlUnwind** is called specifying the address of the target frame to unwind to and the address of the exception-handler-structure as the continuation instruction address.

The **Windows NT** exception dispatcher performs a walk backwards through the call frames when an exception is raised. If it encounters a frame with a handler that has the low-order bit set, it knows that this is not really the address of a handler, but rather the address of an exception handler list for the frame. It calls each handler in the list, one after the other, exactly in the same manner as OS/2, thus implementating exactly the exception dispatching semantics of OS/2.

The function **RtlUnwind** also performs a walk backwards through the call frames when an unwind operation is initiated. This function also recognizes that frames containing a handler with the low-order bit set really point to a list of OS/2 style exception handlers. If the target of the unwind is a frame that contains an exception handler list, then it is known that the continuation address is really the address of an exception-handler-structure that is the target of the unwind and that control is to be returned to the caller of unwind. This implements exactly the same unwind semantics as OS/2.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, May 22, 1989

Revision 1.1, June 2, 1989

1. Major update to include first draft comments.
2. Added section on the implementation of OS/2 exception handling on top of the **Windows NT** capabilities.

Revision 1.2, June 6, 1989

1. Minor corrections of typos.
2. Deleted parameter to illegal instruction, privileged instruction, and invalid lock sequence exceptions to make them more portable.
3. Changed the type name of the context record to match the definition of thread context in the process structure.

Revision 1.3, August 4, 1989

1. Changed the exception dispatch sequence to include a second call to the debugger just before calling the system service emulation subsystem.
2. Changed the name of the *RECURSIVE_CALL* flag to *NESTED_CALL*.
3. Changed the definition of **NtLastChance** so that the function returns an access violation status if the exception or context record are not accessible to the calling process.

Revision 1.4, August 15, 1989

1. Changed names of exception flags to include a leading "EXCEPTION_" tag.
2. Changed field names in the context record to reflect the actual implementation which uses the context record as a call frame.
3. Changed the name of the exception dispatcher to a private internal name and added stack limit parameters.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Changed the exception disposition values from manifest constants to an enumerated type.

5. The exception STATUS_INVALID_DISPOSITION is raised if an invalid disposition value.

6. Change name of NtContinueExecution to NtContinue and add a boolean parameter to specify whether a test alert should be executed.

7. The registers f0, f1, and r0 are no longer saved in the context record.

Revision 1.5, November 6, 1989

1. Delete stack limit arguments from exception dispatcher routine.

2. Change name of collided unwind status code from ExceptionNestedUnwind to ExceptionCollidedUnwind.

3. Change name of exception dispatcher from RtlpDispatchException to RtlDispatchException.

4. Change name of unwind routine from NtUnwind to RtlUnwind.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License