

# **INtime Exception Handling**

## **Detecting and Handling Real-time Exceptions**

October, 2004

TenAsys Corporation

This document is protected by US and international copyright laws.

INtime and iRMX are registered trademarks of TenAsys Corporation.

All trademarks, registered trademarks and brand names are the property of their respective owners.

Products described in this document may be protected by patents, or pending patent applications.

Information regarding products other than those from TenAsys has been compiled from available manufacturers' material. TenAsys cannot be held responsible for inaccuracies in such material.

TenAsys reserves the right to make changes to specifications and product descriptions at any time, without notice. Contact your local TenAsys sales office or distributor to obtain the latest specifications and product descriptions.



PHONE: +1 (503) 748-4720

FAX: +1 (503) 748-4730

1600 NW Compton Drive  
Suite 104  
Beaverton, OR 97006  
USA

EMAIL: [info@tenasys.com](mailto:info@tenasys.com)

WEB: [www.tenasys.com](http://www.tenasys.com)

## Types of Exceptions

Four types of exceptions may occur during the execution of an INtime real-time application, they are: programming, environmental, floating-point, and hardware exceptions.

- 1) Programming exceptions result from program errors (i.e. invalid parameters) made as the result of an INtime system call. It is the responsibility of the programmer to check for an error return from each system call.
- 2) Environmental exceptions also result from an INtime system call – but they are caused by running out of a system resource (e.g. real-time memory or objects), or as the result of a time-out while waiting for some event to occur. Like programming exception errors, it is the responsibility of the programmer to check for an error return from each system call.
- 3) Floating-point exceptions are detected in the floating point unit (FPU) and result from an FPU instruction that would give invalid results (e.g., divide-by-zero, overflow, underflow). By default, FPU exceptions are handled internally by the FPU, so explicit steps must be taken in order for the application to receive notification and handle this type of exception.
- 4) An exception detected by the system hardware (e.g., page fault, general protection fault) is always considered fatal to the executing real-time thread. The exception may also be fatal to the parent real-time process, and affect other real-time processes. The INtime kernel provides a mechanism for hardware exception notification, and a method for specifying the fate of an offending real-time thread and its parent process.

## Programming and Environmental Exceptions

When successful, an INtime system call returns a status code indicating there were no programming or environmental exceptions detected during the processing of the call. The specific status code used to indicate a successful call is dependent on the type of system call made.

The following system call attempts to create a real-time semaphore. This call could return `BAD_RTHANDLE` as a result of either a programming error (e.g., invalid parameter) or an environmental error (e.g., out of memory or out of real-time objects). If the system call fails, a call to `GetLastRtError()` will return an error code indicating the exact cause of the failure.

```
if ( sem_handle = CreateRtSemaphore(0,1,FIFO_QUEUEING) == BAD_RTHANDLE) {
    printf("Failed with error (%04x) attempting to create semaphore.\n",
        GetLastRtError() );
    // Exit or take other appropriate action here...
}
```

Note that `GetLastRtError()` returns the most recent failure for the calling real-time thread. It should always be called immediately after a system call has failed (before the next system call) in order to insure an accurate diagnosis of the failure is retrieved.

## Floating-Point Exceptions

Floating-point exceptions are, by default, handled internally by the FPU. In order to receive an FPU exception notification, an INtime real-time process must:

- 1) Contain an exception handler function that will be called when the FPU exception occurs.
- 2) Call the INtime `SetRtExceptionHandler()` function to assign the exception handler to the real-time process.
- 3) Write a control word to the FPU instructing it to pass control to the software exception handler when an exception occurs. This must be done by every real-time thread that needs to pass control to a handler for FPU exceptions.

Note that the SSE/SSE2 floating point instructions are also supported by this mechanism, although support for those instruction it is not illustrated below. For a complete example illustrating how to handle all floating-point exceptions see the *fpexcept* project included with the sample INtime projects.

### An Example of an FPU Exception Handler

In the following example the actual status word from the FPU is supplied in the first parameter. The low-order 7 bits of the status word indicate the type of FPU exception that was generated, as follows:

Bit 0 = 1	Invalid Operation
Bit 1 = 1	Denormalized Operand
Bit 2 = 1	Zero Divide
Bit 3 = 1	Overflow
Bit 4 = 1	Underflow
Bit 5 = 1	Precision
Bit 6 = 1	FPU Stack Fault

```
__EXCEPTION
void ExceptionHandler(DWORD fpstatus, WORD reserved, WORD paramnum, WORD excep)
{
    __EXCEPTION_PROLOG()

    printf("The exception handler was called with exception code:
           %X and FP status: %X.\n", excep, (BYTE)fpstatus & 0x7f);

    __EXCEPTION_RETURN()
}
```

## Assigning a Handler for an FPU Exception

In the following code, the members of an `EXCEPTION` structure are setup to point to our exception handler, and to the code segment for this process. The value assigned to the `ExceptionMode` structure member determines which exceptions are to be handled.

```
EXCEPTION ExceptStruct;    // declaration for exception structure

// at a later point in the code..

ExceptStruct.ExceptionHandlerPtr = (LPPROC)ExceptionHandler;
__asm { mov ExceptStruct.ExceptionHandlerPtr_seg, cs }
ExceptStruct.ExceptionMode = 1;

if (!SetRtExceptionHandler(&ExceptStruct)) {
    printf("Cannot set exception handler, error %X.\n", GetLastError());
    ExitRtProcess();
}
```

## Enabling Exceptions at the FPU

The following code fragment instructs the FPU to pass control to a software exception handler. Every real-time thread needing to handle FPU exceptions must execute this code.

```
WORD FpControlWord;
#define EM_NONE    ~(EM_INVALID | EM_DENORMAL | EM_ZERODIVIDE | \
                    EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT)

// at a later point in the code..

__asm {                                // Enable FPU exceptions to occur in
software                                software
    fstcw  FpControlWord
    and    FpControlWord, EM_NONE    // clear exception masks
    fldcw  FpControlWord
}
```

## Hardware Exceptions

A hardware exception can be generated for a variety of reasons, but usually are the result of accessing an invalid memory location (e.g., page fault or stack fault). These faults are generally caused by unexpected runtime conditions, and are considered fatal to the continuation of the offending thread. Real-time applications have the option of including a recovery mechanism so they can be notified of the faulting condition and determine what action to take as a result of a hardware exception.

When a hardware exception occurs during the execution of an INtime real-time thread, error information is automatically sent to one or more data mailboxes. A global data mailbox, cataloged in the INtime kernel root directory with the object name `HW_FAULT_MBX`, is automatically created and maintained by the system and exists for the purpose of monitoring hardware exceptions. Each real-time

process can optionally create a local data mailbox, which must be named `HW_FAULT_MBX`, and catalog it into their local object directory, for monitoring process-specific hardware exceptions.

The default action taken by the kernel, when a hardware exception occurs, is to suspend the real-time thread that caused the exception. Although a hardware exception is generally considered fatal to the executing thread, other threads in the same process remain eligible to execute and are not suspended. When a hardware exception occurs, any thread (within the same or another real-time process) that is monitoring the global `HW_FAULT_MBX` can send a notification and/or take other appropriate actions.

Threads that monitor their local `HW_FAULT_MBX` data mailbox will only be notified of hardware exceptions that occur within their own process space. That is, by creating a local `HW_FAULT_MBX` data mailbox one can limit notifications of hardware exceptions to those that are specific to a process. By monitoring the global `HW_FAULT_MBX` data mailbox that resides in the root object directory applications can be notified of hardware exceptions generated by all real-time processes.

The exception message sent to the mailbox is a structure of the type `HWEXCEPTIONMSG`. This structure contains handles identifying the offending thread and its associated process, an error code identifying the cause of the hardware exception, and the instruction pointer for the exception incidence.

## Responding to a Hardware Exception

In the following code fragment, a thread waits on the global `HW_FAULT_MBX`, displays the exception code, and determines whether the parent real-time process was deleted.

```
If (hw_mbx_h = LookupRtHandle(hRootProcess, "HW_FAULT_MBX", WAIT_FOREVER))
    == BAD_RTHANDLE) {
    Fail("Cannot get handle for HW_FAULT_MBX.");
}

if ( !ReceiveRtData(hw_mbx_h, &msg, WAIT_FOREVER) ) {
    Fail("Failed getting data from HW_FAULT_MBX.");
}

printf("Hardware exception type %x occurred.\n", msg.exception);

// Using information from mailbox, determine if parent process was deleted..

if (GetRtHandleType(msg.process) == INVALID_TYPE) {
    printf("The parent process was deleted.\n");
    // Take appropriate action..
}
```

`GetRtHandleType()` (or `ntxGetType()`) can be used to determine the current state of the offending thread and its process by inspecting the returned handle type according to the following logic:

- If the thread and process are valid, then the offending thread was suspended.
- If the process is valid but the thread is not, then the offending thread was deleted.
- If both the thread and the process are invalid, then the offending thread's process was deleted.

## Changing the System Response to a Hardware Exception

Although, the default response is to suspend a real-time thread when a hardware exception occurs, it is possible to change this behavior. It may be appropriate to delete the parent process, for example, if further execution by sibling threads would be inappropriate. After a call to `SetRtExceptionHandler()`, with the `ExceptionMode` value (within the `EXCEPTION` structure) set to twelve, any newly created real-time process will be deleted if a hardware exception occurs during the execution of one of its threads.

Following are the system responses to a hardware exception that can be set by `SetRtExceptionHandler()`:

- 12) Change system-wide hardware trap handlers to delete the offending process.
- 13) Change system-wide hardware trap handlers to delete the offending thread.
- 14) Change system-wide hardware trap handlers to suspend the offending thread.

Note that this behavior specification is system-wide, not process-specific.

For a complete example illustrating how to monitor hardware exceptions see the *ntrobust* project included with the sample INtime projects.